МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

О.К. Тесленко, І.П. Дробязко

СИСТЕМНЕ ПРОГРАМУВАННЯ

КУРСОВА РОБОТА

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського як навчальний посібник для здобувачів ступеня бакалавра за освітньою програмою «Системне програмування і спеціалізовані комп'ютерні системи» спеціальності 123 «Комп'ютерна інженерія»

Київ

«КПІ ім. Ігоря Сікорського»

2021

Рецензенти: *Жабін В.І.*, д-р техн. наук, проф.
Сімоненко В.П., д-р техн. наук, проф.

Відповідальний редактор *Тарасенко В. П.*, д-р техн. наук, проф.

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського (протокол № 5 від 14.01.2021 р.) за поданням Вченої ради факультету прикладної математики (протокол № 5 від 14.12.2020 р.)

Електронне мережне навчальне видання

Тесленко Олександр Кирилович, канд. техн. наук, доц.
Дробязко Ірина Павлівна, ст. викл.

СИСТЕМНЕ ПРОГРАМУВАННЯ

Системне програмування: Курсова робота [Електронний ресурс]: навч. посібн. для студ. спеціальності 123 «Комп'ютерна інженерія» / О.К. Тесленко, І.П. Дробязко; КПІ ім. Ігоря Сікорського. – Електронні текстові дані (1 файл: 1,8 Мбайт). – Київ: КПІ ім. Ігоря Сікорського, 2021. – 162 с.

Навчальний посібник розроблено для виконання курсової роботи з дисципліни «Системне програмування», а саме створення компілятора для програм, що написані мовою Асемблера. Посібник містить вимоги до роботи та оформлення її результатів, опис методики виконання завдання з прикладами та додаткові інформаційні матеріали. Навчальне видання призначене для студентів, які навчаються за спеціальністю 123 «Комп'ютерна інженерія» факультету прикладної математики КПІ ім. Ігоря Сікорського

ЗМІСТ

ВСТУП

Дисципліна «Системне програмування» є спеціальною дисципліною циклу професійної та практичної підготовки бакалаврів за спеціальністю 123 «Комп'ютерна інженерія», яка ознайомлює студентів з проблемами, які стоять перед системним програмуванням, і основними напрямками їх вирішення. Одним з важливих видів індивідуальних завдань, що передбачені навчальною програмою дисципліни «Системне програмування» та виконуються студентами при її вивченні, є курсова робота (КР).

КР є творчим рішенням конкретної задачі, а саме розробки компілятора програм мовою Асемблера, виконуваним студентами самостійно під керівництвом викладача згідно із завданням, на основі набутих знань та умінь з даної дисципліни та суміжних дисциплін: «Основи програмування», «Структури даних та алгоритми», «Комп'ютерна логіка. Комп'ютерна арифметика», «Об'єктно-орієнтоване програмування». Виконання курсової роботи сприяє розширенню та поглибленню отриманих теоретичних знань щодо архітектури процесорів, машинної мови та машинно-орієнтованої мови програмування Асемблера та ін., формуванню вмінь їх практичного використання, самостійного вирішення відповідних інженерних завдань програмним шляхом, розвитку навичок тестування та документування розробленого програмного продукту.

Посібник містить загальну постановку завдання та зміст індивідуального завдання, рекомендації щодо методики реалізації завдання з прикладами та додатковими інформаційними матеріалами, вимоги до виконання завдання, оформлення звіту та захисту курсової роботи тощо.

1. МЕТА ТА ЗАВДАННЯ КУРСОВОЇ РОБОТИ

Метою курсової роботи з дисципліни «Системне програмування» є закріплення та поглиблення знань щодо мови Асемблера та її взаємозв'язку з архітектурою процесора, набуття практичних навичок у реалізації компіляторів програм мовою Асемблера.

Завданням даної курсової роботи є розробка компілятора програм, написаних мовою Асемблера. В результаті виконання **курсової роботи** студент має:

ЗНАТИ:

- особливості архітектури процесорів фірми Intel, внутрішнє представлення та формати інструкцій і даних; відповідність машинно-орієнтованої мови програмування Асемблера машинній мові та ін.;
- загальну методику побудови компіляторів (лексичний аналіз, синтаксичний аналіз, обробка лексем, формування коду, виявлення помилок тощо);

УМІТИ:

- самостійно працювати з навчально-методичною, технічною, періодичною літературою й іншими інформаційними джерелами за тематикою розробки;
- практично використовувати отримані знання для самостійного вирішення завдання програмним шляхом і побудови компілятора;
- аналізувати та алгоритмізувати поставлену задачу, обґрунтовувати структурну організацію проекту, оцінювати

доцільність застосування обраних програмних засобів для створення програмного продукту;

– документувати новостворений програмний продукт, давати опис структури програми та схеми взаємодії її складових;
– розробляти тестові модулі та використовувати їх для тестування розробленого програмного продукту;

НАПРАЦЮВАТИ ДОСВІД:

- роботи в різних середовищах розробки та відлагодження програм;
- створення завершених програмних продуктів.

2. ЗАВДАННЯ НА КУРСОВУ РОБОТУ

Тема курсової роботи – Розробка компілятора для програм, що написані мовою Асемблера.

Основним завданням і результатом роботи компілятора повинно бути створення для програм мовою Асемблера текстового файла лістинга (розширення .lst), подібного до файла лістинга компілятора TASM.

Враховуючи, що створення компілятора є трудомістким процесом, який потребує значних витрат часу, у варіантах завдань на курсову роботу надаються суттєві обмеження на перелік допустимих машинних інструкцій, режимів адресації даних і команд та допустимих директив. Будь-який із запропонованих студентам варіантів індивідуальних завдань фактично вказує на підмножину стандартної мови Асемблера процесорів фірми Intel. Тим не менш, цієї підмножини достатньо для забезпечення мети розробки.

Нижче наведено приклад індивідуального завдання на розробку компілятора з вищезазначеними обмеженнями вимог до компілятора.

Приклад індивідуального завдання

Ідентифікатори

Містять великі та малі букви латинського алфавіту і цифри. Починаються з букви. Великі та малі букви не відрізняються.

Довжина ідентифікаторів – не більше 8 символів.

Константи

Шістнадцяткові, десяткові, двійкові та текстові константи.

Директиви

END, SEGMENT без операндів, ENDS, ASSUME.

DB, DW, DD з одним операндом – константою (рядкові константи тільки для DB).

Розрядність даних та адрес

16-розрядні дані та зміщення у сегменті, у випадку 32-розрядних даних та 32-розрядних зміщень генеруються відповідні префікси зміни розрядності.

Адресація операндів пам'яті

Індексна адресація (Val1[si],Val2[di],Val3[eax],Val4[edi] і т.п.).

Заміна сегментів

Префікси заміни сегментів можуть задаватись явно, а при необхідності автоматично генеруються компілятором.

Машинні команди

Cli

Inc **reg**

Dec **mem**

Add **reg, reg**

Cmp **reg, mem**

Xor **mem, reg**

Mov **reg, imm**

Or **mem, imm**

Jb

Jmp (внутрішньосегментна відносна адресація),

де **reg –** 8, 16 або 32-розрядні регістри загального призначення (РЗП);
mem – адреса операнда в пам'яті;
imm - 8, 16 або 32-розрядні безпосередні дані (константи).

Завдання на виконання КР студенти отримують на початку семестра, в якому передбачена курсова робота. У завданні сформульована загальна постановка завдання, індивідуальне завдання, визначені основні вихідні дані, вимоги до роботи та очікувані результати, а також календарний план-графік виконання курсової роботи. В окремих випадках, пункти завдання (або завдання в цілому) можуть бути змінені студентом за згодою викладача.

3. СКЛАД, ОБСЯГ І СТРУКТУРА КУРСОВОЇ РОБОТИ

3.1. Вихідні дані та загальні вимоги до розробки

1. *Вихідні дані* компілятора – розроблений студентом файл з довільною програмою мовою Асемблера, яка складена у відповідності з обмеженнями індивідуального варіанта курсової роботи (далі – тестова програма).
2. *Результатом роботи* компілятора повинно бути створення текстового файла лістинга (розширення .lst).

Формат файла лістинга має співпадати з форматом файла лістинга компілятора TASM. Таблиця символів у файлі лістинга може бути представлена у довільному форматі.

3. *Імена початкового асемблерного файла* для обробки компілятором та створюваного файла лістинга повинні задаватися у командному рядку при запуску програми компілятора.
4. *Усі діагностичні повідомлення*, що міститиме файл лістинга, повинні також виводитися на екран монітора. Крім того, на екран

повинна виводитися загальна кількість помилок, виявлених у початковій програмі.

5. *На усі синтаксичні конструкції* (ідентифікатори, константи, директиви, машинні команди, режими адресації тощо), які допускаються в компіляторі TASM, проте виходять за рамки обмежень варіанта курсової роботи, повинно видаватись *діагностичне повідомлення про помилку*.
6. Не рекомендується деталізувати діагностичне повідомлення про помилку. Достатньо вивести коротке повідомлення на будь-які виявлені помилки, наприклад – *Помилка*.
7. При виявленні помилки, на відміну від TASM, аналіз чергового рядка припиняється, а на обох переглядах кількість згенерованих байтів для рядка **дорівнює нулю**.

При здійсненні розробки компілятора повинні бути послідовно вирішені наступні задачі:

1. Створення тестових програм мовою Асемблера, що відповідають вимогам індивідуального завдання та дозволяють перевірити коректність роботи компілятора:
   - програми без помилок і з обмеженнями індивідуального завдання;
   - створення аналогічної тестової програми та отримання для неї за допомогою компілятора TASM файла лістинга для подальшого його порівняння з результатами роботи розробленого компілятора.
2. Створення лексичного аналізатора програми мовою Асемблера.

3. Створення програми 1-го перегляду (формування таблиці ідентифікаторів, визначення кількості байтів, які будуть формуватися за кожною інструкцією).
4. Створення програми 2-го перегляду (генерування команд та даних, формування файла лістинга).

3.2. Вимоги до мови програмування

Мова програмування для створення компілятора обирається студентом самостійно та узгоджується з викладачем.

3.3. Вимоги до тестових файлів

Тестовий файл є початковою програмою мовою Асемблера зі структурою, яка відповідає вимогам до початкових програм – мати два або більше (у відповідності до конкретного завдання) логічних сегментів та закінчуватись директивою END.

Сукупність рядків в межах логічних сегментів повинна відповідати лише правилам синтаксису відповідних директив чи машинних інструкцій Асемблера згідно із завданням. **Реалізація того чи іншого алгоритму у такій початковій програмі не передбачається і не рекомендується. Наприклад, команда передачі управління за умовою може бути у будь-якому місці тесту, а не тільки після команди порівняння.**

У тестовій початковій програмі необхідно забезпечити перевірку виконання кожного із елементів конкретного завдання, зокрема, кожна із заданих в завданні директив і машинних інструкцій повинна зустрічатись принаймні один раз.

Вимоги до машинних інструкцій тестової програми:

- використовувати кожен із режимів адресації, передбачених у завданні;
- забезпечити звернення до кожного із заданих типів (розрядностей) даних у пам’яті;
- використовувати хоча б один із регістрів даних заданої розрядності;
- задати принаймні одну константу заданого формату (шістнадцяткова, десяткова, двійкова і т.п.);
- задати принаймні одну константу (абсолютний вираз) заданої розрядності;
- команди передачі управління за умовою та внутрішньо-сегментні безумовні (внутрішньо-сегментні виклики процедур) задати принаймні два рази – з посиланням вперед та посиланням назад.

Тестування пункту «*префікси заміни сегментів при необхідності автоматично генеруються транслятором*» здійснювати наступним чином:

- якщо у завданні вказано, що програма повинна містити лише один сегмент даних і один сегмент кодів, тоді у сегменті кодів необхідно задати директиву визначення пам’яті (DB, DW чи DD) з обов’язковою вказівкою імені та забезпечити звертання до цих даних принаймні в одній із машинних команд;
- якщо програма може містити принаймні два логічних сегменти даних, тоді один із них у директиві ASSUME необхідно зв’язати з сегментним регістром, який не використовується за

замовчуванням, та забезпечити відповідне звернення до даних із цього сегменту.

УВАГА! Для перевірки повноти тестової програми необхідно пересвідчитись у наявності перевірки кожного пункту завдання у тестовому файлі.

Як вищезазначено, на першому етапі виконання роботи студенти повинні підготувати два тестові файли: файл тестової програми для розроблюваного компілятора та модифікований файл для TASM. Суть модифікації полягає у забезпеченні *режимів і умов для трансляції тестової програми* для компілятора TASM. До таких режимів і умов належать наступні:

1) Не залежно від завдання, першою директивою модифікованого тесту повинна бути директива *.486*.
2) Якщо в завданні не задана директива Assume, то у модифікованому тесті ця директива є обов'язковою.
3) Якщо у завданні за замовчуванням задано оброблення 16-розрядних даних та 16-розрядних адрес, то у директивах *Segment* модифікованого тесту необхідно задати операнд *Use16*.
4) Якщо в ідентифікаторах можуть використовуватись *букви українського алфавіту*, то у модифікованому файлі букви українського алфавіту необхідно замінити на букви латинського (транслітерація).

Зразок тестового файла, що створений для **Прикладу індивідуального завдання** (див. п. 2), представлено у Додатку В.

4. МЕТОДИКА РОЗРОБКИ КОМПІЛЯТОРА

4.1. Структурна схема компілятора Асемблера

Теорія та практика створення компіляторів мов програмування досить глибоко та всебічно розвинута та буде вивчатись у відповідній дисципліні. Побудова компіляторів мов високого рівня ґрунтується на використанні формальних граматик, теорії цифрових автоматів та ін. Більшість сучасних компіляторів Асемблера використовують досягнення у побудові компіляторів мов високого рівня. Проте, на жаль, при цьому втрачається прозорість та логічна простота процесу компіляції програм, написаних мовою Асемблера. Використання фактично надлишкових можливостей у деякій мірі відвертає увагу від основоположних проблем, які вирішуються як при створенні мови Асемблера, так і при трансляції програм мовою Асемблера. Тому будемо розглядати побудову Асемблера за простою класичною двохпереглядною схемою з використанням прямих методів трансляції.

Перегляд – послідовна обробка символів файла початкової програми. Послідовність двох рівнів – послідовність рядків програми та послідовність символів у рядках. У випадку класичної двохпереглядної схеми Асемблера файл початкової програми переглядається 2 рази.

Прямий метод трансляції – почергове порівняння рядка програми або частини рядка з можливими синтаксичними конструкціями мови Асемблера. Звичайно, прямі методи трансляції доцільно використовувати лише при незначній кількості можливих синтаксичних конструкцій, що притаманно мові Асемблера.

Структурна схема компілятора за класичною двохпереглядною схемою приведена на рис.4.1. Будемо вважати, що початкова програма представлена деяким файлом у символьному форматі, наприклад, у коді

ASCII з розширенням буквами українського алфавіту, тобто код кожного символу програми міститься в одному байті. Для спрощення будемо вважати, що будь-яка інструкція програми повністю розміщується в одному рядку. Компілятор на кожному перегляді послідовно читає рядки програми, а прочитавши рядок – послідовно переглядає символи рядка. Коли компілятор розпізнає директиву *END*, він закінчує роботу з чергового перегляду.

Розпізнавання, з точки зору програми, – це передавання управління на ту частину програми (або процедуру), де виконується обробка відповідної частини рядка. У загальному випадку, для такого передавання управління необхідно виконати досить складний аналіз як поточних символів рядка, так і структур даних, які створені при обробці попередніх рядків або при створенні самого компілятора.

Задачі, що вирішуються на переглядах

На ***першому перегляді*** вирішуються наступні задачі:

1. *Розподіл пам'яті*, тобто визначення зміщень команд та даних у відповідних логічних сегментах.

Для вирішення цієї задачі мова Асемблера повинна давати можливість визначення кількості байтів, які будуть генеруватись за кожним рядком програми. Визначення їх вмісту на першому перегляді у класичній двохпереглядній схемі не передбачається.

2. *Формування таблиці ідентифікаторів*, які визначаються користувачем.

До таких ідентифікаторів належать:

- *мітки* (символічні адреси команд, тобто зміщення першого байта команди у відповідному логічному сегменті);

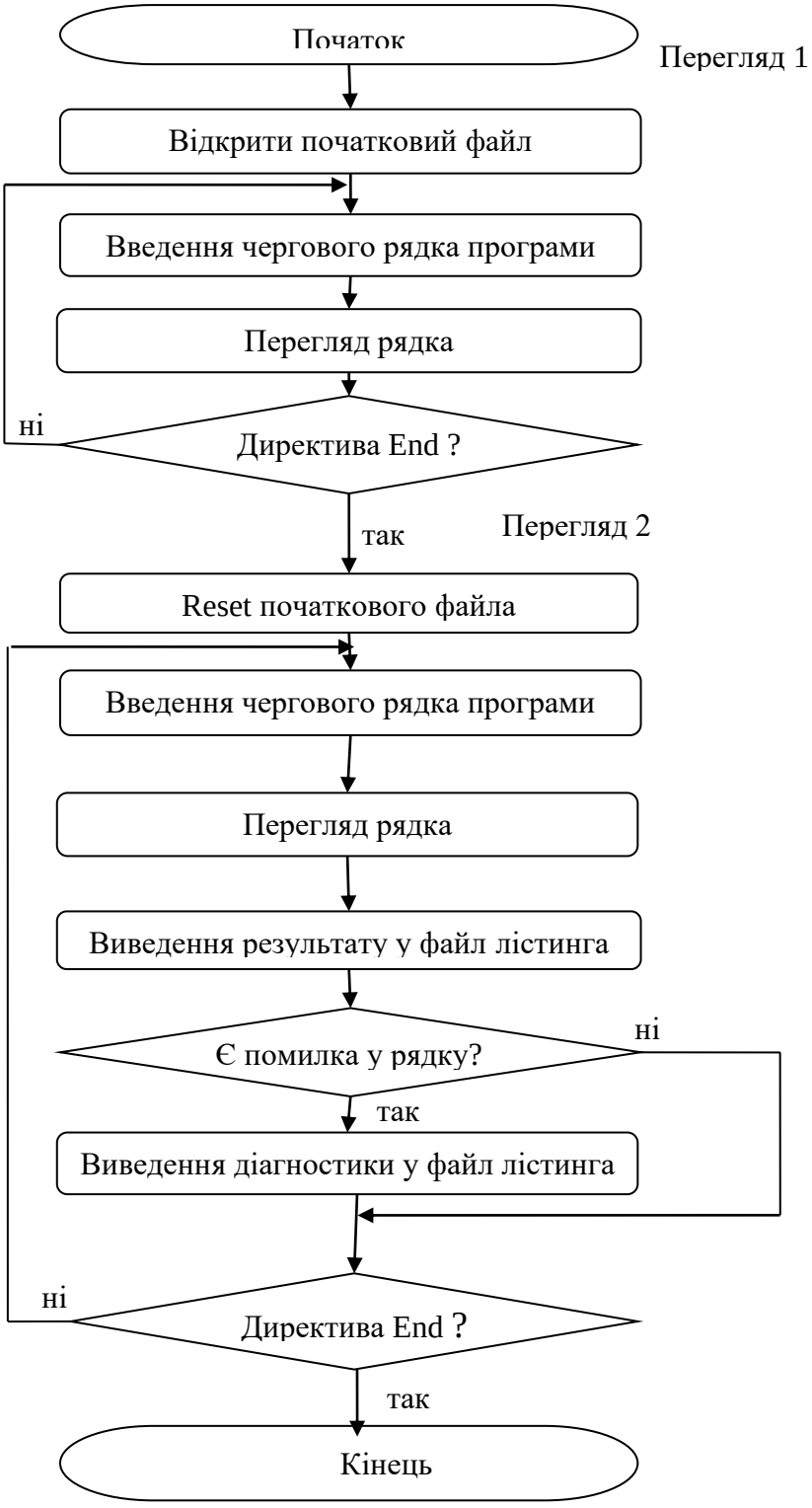


Рисунок 4.1 – Структурна схема компілятора

- *імена у директивах визначення пам'яті* (символічні адреси даних – зміщення молодшого байта даних у відповідному логічному сегменті);
- *імена процедур* (символічні адреси процедур – зміщення першого байта коду процедури у відповідному логічному сегменті);
- *символічне позначення* відповідних послідовностей символів у директиві EQU;
- *символічне позначення констант* у директиві = ;
- *імена макрокоманд*;
- *імена програмних об'єктів* у директиві *Extrn*.

3. *Виявлення помилки* – повторне визначення одного й того ж ідентифікатора.
4. *Формування таблиці логічних сегментів*.
5. *Формування бібліотеки макровизначень*.
6. *Формування макророзширень для макрокоманд*.

На ***другому перегляді*** вирішуються наступні задачі:

1. *Генерування вмісту байтів команд та байтів даних* за рядками початкової програми.
2. *Формування файла лістинга* з додаванням таблиці логічних сегментів і таблиці ідентифікаторів, визначених користувачем (з першого перегляду).
3. *Формування повідомлень про помилки* у файлі лістинга, також про помилки повторного визначення ідентифікаторів, які виявлені на першому перегляді.

Дії та структури даних Асемблера на кожному з переглядів

Для вирішення вказаних задач як на першому, так і на другому перегляді над кожним рядком програми виконуються наступні дії:

- лексичний аналіз;
- *визначення структури речення програми* (визначення полів машинної інструкції або директиви) – елементи синтаксичного аналізу;
- елементи граматичного аналізу, в результаті якого безпосередньо виконуються задачі, що покладаються на відповідний перегляд.

Оскільки перелік дій компілятора для першого і другого перегляду співпадають, далі при розгляді відповідних дій увага буде акцентуватись на особливостях кожного з переглядів.

Під час роботи компілятора використовується та формується ряд таблиць:

1) *Таблиця зарезервованих ідентифікаторів* (ключових слів мови Асемблера), до яких належать: мнемокоди машинних інструкцій і директив Асемблера, імена регістрів, імена операторів та операцій, імена операндів в окремих директивах та ін. Ця таблиця формується при розробці компілятора та є незмінною.
2) *Таблиця ідентифікаторів, які визначив користувач* у програмі (імена, мітки і т.п.). Ця таблиця формується на першому перегляді, на другому перегляді лише використовується і не доповнюється (можлива модифікація у випадку визначення абсолютного значення директивою =).
3) *Таблиця логічних сегментів*. Ця таблиця формується на першому перегляді та частково модифікується на другому перегляді.

4) *Таблиця назначень сегментних регістрів*. Ця таблиця заповнюється значенням NOTHING на початку переглядів та модифікується при обробці директиви *Assume* в обох переглядах.
5) *Таблиця лексем*. Ця таблиця формується для кожного рядка програми на обох переглядах та після обробки чергового рядка обнуляється.
6) *Таблиця структури рядка програми*. Ця таблиця формується для кожного рядка програми на обох переглядах та після обробки чергового рядка обнуляється.
7) *Бібліотека макровизначень*. Ця бібліотека формується на першому перегляді, а на другому перегляді лише використовується і не змінюється.
8) *Таблиця формальних параметрів макрокоманди*. Ця таблиця формується при обробці директиви *Macro*, а при обробці директиви *Endm* обнуляється.
9) *Таблиця фактичних параметрів макрокоманди*. Ця таблиця формується при створенні макророзширення, а при обробці директиви *Endm* обнуляється.

Із вищезазначеного випливає, що при побудові компілятора Асемблера важливе значення має створення процедур читання та записування даних у таблиці.

4.2 Лексичний аналіз

В результаті лексичного аналізу формується *таблиця лексем* чергового рядка програми.

Лексема – один або більше символів у рядку, які Асемблером розглядаються як єдиний об’єкт. Лексеми – “неподільні атоми” мови програмування. У зв’язку з цим, часто використовується термін

«термінальний символ» мови програмування. В мові Асемблера розрізняють наступні лексеми:

- *Ідентифікатори* – послідовності букв та цифр, які розпочинаються з букви.

До букв відносяться малі та великі букви відповідних алфавітів. В TASM таким алфавітом є латинський алфавіт, а також символи _, @, ?, $. Загальноприйнято не розрізняти малі та великі букви. В той же час, малі та великі букви в ASCII мають різні коди. Тому при обробці ідентифікаторів компілятор перетворює усі малі букви у великі. Якщо компілятор розробляється, наприклад, в Україні, то доречно в якості букв ідентифікаторів ввести і букви українського алфавіту. Латинський та український алфавіти мають багато однакових за зовнішнім виглядом букв, але вони мають різні коди ASCII (наприклад a, e, p, x та ін.). Для уникнення потенційних помилок, пов’язаних з неправильним переключенням алфавіту при введенні початкових програм, компілятор при обробці ідентифікаторів повинен для однакових за зовнішнім виглядом букв латинського та українського алфавітів вибрати один і той самий код. Довжина ідентифікатора практично не обмежується, але значущими є перші 32 символи.

- *Числові константи* – послідовності цифр та деяких букв, які розпочинаються з цифри.

Розрізняють двійкові, вісімкові, десяткові та шістнадцяткові константи. В якості допустимих букв числової константи є великі та малі (вони, як і в ідентифікаторах, не розрізняються) букви a, b, c, d, e, f та букви основи систем числення: b, o, q, d, h.

Двійкові константи складаються із цифр 0 та 1 та обов’язково закінчуються буквою *b*.

Вісімкові константи складаються із цифр від 0 до 7 і обов'язково закінчуються буквою *o* або *q*.

Десяткові константи складаються із цифр від 0 до 9 та можуть закінчуватись буквою *d* (допускається її відсутність наприкінці). Ознакою закінчення десяткової константи у цьому випадку є поява символу, який не входить до переліку цифр від 0 до 9.

Шістнадцяткові константи складаються із цифр від 0 до 9 і букв a, b, c, d, e, f та обов'язково закінчуються буквою *h*. Якщо першим символом шістнадцяткової константи є буква, то спершу необхідно поставити цифру 0, щоб компілятор зміг відрізнити ідентифікатор від числової константи.

- *Текстові константи* – послідовність довільних символів, які можна ввести з клавіатури та які розпочинаються і закінчуються символом ‘ або “. **Символи у текстових константах ніяким перетворенням не підлягають**.
- *Розподільники лексем* – символи пробілу, табуляції та крапка з комою.
- *Односимвольні лексеми* – всі інші символи, які входять до алфавіту мови Асемблера.

Розподільник між двома лексемами є обов'язковим, якщо конкатенація (пристиковування) цих лексем є також лексемою.

Таблиця лексем може мати, наприклад, наступну структуру (табл. 4.1):

Таблиця 4.1 – Структура таблиці лексем

№ п/п	Лексема	Довжина лексеми у символах	Тип лексеми

Наприклад, для рядка

*@30: Mov eax, dword ptr gs:Aray1[ebx+esi*4+40h]; формування покажчика*

буде створена наступна таблиця із 19 лексем (табл. 4.2):

Таблиця 4.2 – Приклад 1 заповнення таблиці лексем

№ п/п	Лексема	Довжина лексеми у символах	Тип лексеми
1.	@30	3	Ідентифікатор користувача або не визначений
2.	:	1	односимвольна
3.	MOV	3	Ідентифікатор мнемокоду машинної інструкції
4.	EAX	3	Ідентифікатор 32- розрядного регістра даних
5	,	1	односимвольна
6	DWORD	5	Ідентифікатор тип 4
7.	PTR	3	Ідентифікатор оператора визначення типу
8	GS	2	Ідентифікатор сегментного регістра
9	:	1	односимвольна
10	ARAY1	5	Ідентифікатор користувача або не визначений
11	[	1	односимвольна
12	EBX	3	Ідентифікатор 32- розрядного регістра адрес
13	+	1	односимвольна
14	ESI		Ідентифікатор 32- розрядного регістра адрес
15	*	1	односимвольна
16	4	1	Десяткова константа
17	+	1	односимвольна
18	40H	3	Шістнадцяткова константа
19	]	1	односимвольна

Для рядка

My_data db 10100b, ’зв’язок’

буде створена інша таблиця лексем (табл. 4.3).

Таблиця 4.3 – Приклад 2 заповнення таблиці лексем

№ п/п	Лексема	Довжина лексеми у символах	Тип лексеми
1	My_data	7	Ідентифікатор користувача або не визначений
2	DB	2	Ідентифікатор директиви даних тип 1
3	10100B	6	Двійкова константа
4	,	1	односимвольна
5	зв’язок	7	Текстова константа

Як результат формування таблиці лексем із подальшого розгляду вилучаються всі розподільники та коментарі.

Характерною помилкою, що виявляється при лексичному аналізі, є недопустимий символ, тобто поява у реченні за межами текстової константи символу, який можна ввести з клавіатури, але який не належить до алфавіту мови Асемблера.

4.3 Елементи синтаксичного аналізу

Визначення структури речення програми

Після лексичного аналізу перелік допустимих послідовностей лексем у можливій структурі речень в таблиці лексем може бути зведений до наступних:

<pusto>

<label>:

<mnem>

<mnem> {<op>}

<label>: <mnem>

<name> <mnem>

<label>: <mnem> {<op>}

<name> <mnem> {<op>}

де

<pusto> – таблиця, в якій відсутні лексеми (у випадку порожніх речень та речень коментарів);

<label> – ідентифікатор мітки в машинній інструкції;

<mnem> – ідентифікатор директиви, машинної інструкції чи макрокоманди або послідовність двох ідентифікаторів, одним із яких є префікс повторення, а другим – мнемоніка рядкової команди;

<name> – ідентифікатор імені в директиві;

{<op>} – один або більше операндів, які у більшості випадків є послідовністю лексем, розділених символом коми.

Відкидаючи випадок порожньої таблиці лексем, можна зробити наступні висновки щодо наведених послідовностей. По-перше, першою лексемою рядка завжди є ідентифікатор (якщо інша лексема – це вказує на синтаксичну помилку і подальший аналіз може бути припинено). По-друге, необхідно за допомогою процедури пошуку у таблицях перевірити наявність цього ідентифікатора серед мнемокодів машинних інструкцій, мнемокодів директив і мнемокодів макрокоманд. В результаті такого аналізу створюється таблиця структури речення, де вказуються порядкові номери лексем із таблиці лексем, які належать полю міток, полю мнемокоду та кожному із операндів поля операндів. Таблиця структури речення може мати наступний вигляд (табл. 4.4).

Таблиця 4.4 – Таблиця структури речення

Поле міток (імені)	Поле мнемокоду		1-й операнд		2-й операнд		n-й операнд	
№ лексеми поля	№ першої лексеми поля	Кіл-ть лексем поля	№ першої лексеми операнда	Кіл-ть лексем операнда	№ першої лексеми операнда	Кіл-ть лексем операнда	№ першої лексеми операнда	Кіл-ть лексем операнда

Наявність у полі мнемокоду двох лексем відповідає випадкам використання префіксів повторення для ланцюгових (рядкових) команд.

Відповідно, для двох попередніх прикладів рядків програми (табл. 4.2-4.3 п. 4.2), маємо (табл. 4.5):

Таблиця 4.5 – Приклади структури речень

Поле міток (імені)	Поле мнемокоду		1-й операнд		2-й операнд	
№ лексеми поля	№ першої лексеми поля	Кількість лексем поля	№ першої лексеми операнда	Кількість лексем операнда	№ першої лексеми операнда	Кількість лексем операнда
1	3	1	4	1	6	14
1	2	1	3	1	5	1

Аналіз операндів машинних інструкцій

Машинні інструкції можуть мати наступні операнди:

- *константи* (абсолютні терми) або *вирази над константами* (абсолютні вирази) – використовуються для завдання безпосередніх даних;

- *регістри даних* – у більшості випадків використовуються для формування поля *reg байта modR/m*;
- *перемістимі вирази* – визначення складових зміщення (оператор *offset*) та імені логічного сегмента (оператор *seg*), які використовуються для завдання безпосередніх даних у командах. На відміну від абсолютних виразів, перемістимі вирази формують у командах безпосередні дані, які можуть змінюватися при формуванні виконуваних файлів та при завантаженні програм у пам’ять;
- *адресні вирази* використовуються для завдання адресних частин команд – байтів режиму адресації та зміщення в команді, (як виняток, в командах міжсегментної передачі управління – для завдання логічної адреси). Адресні вирази розпізнаються при наявності в операнді адресних термів (міток або імен, які визначають три складові: ім’я логічного сегменту, зміщення в логічному сегменті та тип) і/або при наявності адресних регістрів. У структурі адресних виразів допускається наявність абсолютних виразів.

Обмежена кількість регістрів даних (8 регістрів), та обмежена кількість різновидів адресних виразів у конкретних індивідуальних завданнях (не більше 2) дозволяє виконувати їх розпізнавання методом прямої трансляції (шляхом перебору можливих варіантів).

Обробка абсолютних виразів

Обробка абсолютних виразів компілятором Асемблера полягає в обчисленні значення виразу. У загальному випадку, існує значна кількість різних абсолютних виразів, тому використовувати прямі методи трансляції

(шляхом перегляду всіх варіантів) практично неможливо. Для обчислення абсолютних виразів використовується алгоритм Дейкстри – використання стеку з пріоритетами. Суть алгоритму полягає у наступному:

1. Якщо чергова лексема виразу є абсолютним термом, його значення безумовно записується в стек даних.
2. Якщо чергова лексема виразу є операцією – порівнюється пріоритет операції на верхівці стеку операцій та пріоритет поточної операції.
3. Якщо пріоритет поточної операції вищий ніж пріоритет операції на верхівці стеку, поточна операція записується в стек операцій. Перейти до п. 1.
4. Якщо пріоритет поточної операції менший або дорівнює пріоритету операції на верхівці стеку, операція читається із стеку операцій. Визначається кількість операндів цієї операції. Відповідна кількість даних читається із стеку даних. Операція, що прочитана із стеку операцій, виконується над прочитаними із стеку даними, а результат записується до стеку даних. Перейти до п.3.
5. Якщо чергова лексема є відкриваючою круглою дужкою, вона безумовно записується до стеку операцій. Вважається, що відкриваюча дужка має найнижчий пріоритет. Перейти до п. 1.
6. Якщо чергова лексема є закриваючою круглою дужкою, вона почергово виштовхує із стеку всі операції до відкриваючої круглої дужки. Потім відкриваюча та закриваюча круглі дужки знищуються. Перейти до п. 1.
7. Якщо всі лексеми вичерпані, а стеки не порожні, почергово виштовхуються із стеку всі операції.

Розглянемо, наприклад, обчислення абсолютного виразу 34+50*(7+10) (рис. 4.2):

№ кроку (поточна лексема)	Стек даних	Стек операцій
1. 34	34	
2. +	34	+
3. 50	50 34	+
4. *	50 34	* +
5. (	50 34	(* +
6. 7	7 50 34	(* +
7. +	7 50 34	+ (* +

№ кроку (поточна лексема)	Стек даних	Стек операцій
8. 10	10	+
	7	(
	50	*
	34	+
9. виштовхується +	17	(
	50	*
	34	+
10. виштовхується (	17	*
	50	+
	34	
11. кінець операнда виштовхується *	850	+
	34	
12. кінець операнда виштовхується +	884	

Рисунок 4.2 – Обчислення виразу 34+50*(7+10)

Наприкінці обчислення абсолютного виразу визначається кількість байтів, які необхідні для збереження значення виразу. Для розглянутого прикладу вона дорівнює 2.

4.4 Елементи граматичного аналізу

Після створення таблиці структури речення, шляхом використання процедури пошуку в таблиці ключових слів мови Асемблера за полем мнемокоду визначається вид речення: машинна інструкція чи директива. Виклад аналізу речень програми доцільно проводити у наступному порядку.

Аналіз основних директив програми на першому та другому переглядах

Обробка директив Segment та Ends

Формування (активізація) чергового рядка таблиці сегментів (табл. 4.6)

Active_seg = номер відповідного рядка таблиці сегментів

здійснюється наступним чином.

Таблиця 4.6 – Таблиця сегментів

Ім’я сегмента	Розрядність за замовчуванням	Поточне зміщення
Code	32	0

На початку перегляду *Active_seg=0*. Будемо вважати, що рядки таблиці сегментів нумеруються з 1.

Поточне зміщення часто називають лічильником адреси Асемблера у поточному логічному сегменті. Мовою Асемблера він позначається символом *$*. При відкритті логічного сегмента на обох переглядах *$=0*. На обох переглядах відбувається пошук ідентифікатора імені сегмента у

таблиці сегментів. Якщо на першому перегляді пошук невдалий, тоді створюється новий рядок у таблиці сегментів з заповненням відповідних полів таблиці за замовчуванням або відповідно до ідентифікаторів операндів. Наступні директиви *Segment* з тим самим ім'ям сегмента лише встановлюють значення *Active_seg*.

Обробка директиви *Ends* полягає в наступному: *Active_seg = 0*.

ЗАУВАЖЕННЯ. Лише директиви *Assume*, *Equ*, *=*, *Macro, Endm*, *Public*, *Extrn* та рядки-коментарі можуть розміщуватися за межами логічного сегмента. У всіх інших рядках програми наявність *Active_seg = 0* розглядається як помилка.

Обробка директив Assume

На початку кожного перегляду в кожен рядок таблиці назначень сегментним регістрам логічних сегментів заноситься ключове слово *Nothing*. Обробка операндів директиви *Assume* полягає у записі в цю таблицю ідентифікатора, який міститься за символом *:* (двокрапка). Розглянемо приклад. На початку кожного з переглядів формується наступна таблиця (табл. 4.7):

Таблиця 4.7 – Таблиця назначень сегментним регістрам

Сегментний регістр	Назначення
CS	Nothing
DS	Nothing
SS	Nothing
ES	Nothing
GS	Nothing
FS	Nothing

Після обробки директиви

Assume CS:code, DS:data

таблиця назначень матиме наступний вигляд (табл. 4.8):

Таблиця 4.8 – Назначення після обробки

Сегментний регістр	Назначення
CS	CODE
DS	DATA
SS	Nothing
ES	Nothing
GS	Nothing
FS	Nothing

Якщо далі в програмі з’явиться директива

Assume CS:code, DS:data2, gs:segs

таблиця матиме наступний вигляд (табл. 4.9):

Таблиця 4.9 – Нове назначення

Сегментний регістр	Назначення
CS	CODE
DS	DATA2
SS	Nothing
ES	Nothing
GS	SEGS
FS	Nothing

ЗАУВАЖЕННЯ. Якщо в завданні відсутня директива *ASSUME*, все ж таки необхідно створювати таблицю назначень сегментним регістрам, де для сегментного регістру *DS* задається ім’я сегменту даних, а для сегментного регістру *CS* – ім’я сегменту кодів.

Обробка директиви ORG <абсолютний вираз>

У рядку *Active_seg* таблиці сегментів у поле “*Поточне зміщення*” записується значення абсолютного виразу, тобто *$ = <абсолютний вираз>*.

Значення *Active_seg = 0* означає розміщення директиви *ORG* поза логічним сегментом, що є помилкою.

Обробка поля міток машинних інструкцій і поля імені директив визначення пам'яті

Якщо поле імені (мітки) не порожнє, воно обробляється *на першому перегляді* за наступним алгоритмом (далі Алгоритм 1):

1) За допомогою процедури пошуку в таблицях виконати пошук ідентифікатора імені (мітки) в таблиці ідентифікаторів, визначених користувачем. При успіху пошуку перейти до п.3.
2) Записати в таблицю визначених користувачем ідентифікаторів мітку (ім'я), задавши значення зміщення з поля "*Поточне зміщення*" активного логічного сегмента; значення сегментної складової – ім'я активного логічного сегмента; значення типу near (*-1*) для мітки або *1, 2, 4, 6, 8, 10* відповідно для директив *db, dw, dd, dp, dq, dt*. Перейти до п.4.
3) Зафіксувати помилку – дублювання імені (мітки).
4) Кінець.

На другому перегляді виконується пошук ідентифікатора у полі мітки (імені) таблиці ідентифікаторів користувача та порівнюється значення зміщення із значенням поля поточного зміщення активного сегмента у таблиці сегментів. При неспівпадінні значень формується діагностичне повідомлення "***Phase error"*** – невідповідність зміщень першого та другого переглядів.

Надалі ідентифікатор користувача, якому в таблиці назначено зміщення, тип та ім'я сегмента будемо називати *адресним термом*.

Обробка інших директив

Обробка директиви Proc та Endp

Ім’я процедури у полі імені обробляється згідно з вищезазначеним Алгоритмом 1. Значення типу визначається ключовим словом у полі операндів директиви: *-1* (якщо *near* або поле операндів порожнє), *-2* (якщо *far*). На другому перегляді значення типу процедури записується у глобальну змінну компілятора, яка використовується для генерації відповідного коду команди *Ret* (*Retn* чи *Retf*). Директива *Endp* на другому перегляді встановлює цю змінну в значення *-1* (*near*). Початкове значення цієї змінної також *-1*. Директива *Endp* на першому перегляді не обробляється.

Обробка директиви Equ

Ім’я у полі імені директиви обробляється згідно з Алгоритмом 1 обробки поля міток і лише на першому перегляді. При цьому, в якості значення записується послідовність символів із поля операндів, а в якості типу – тип *Equ* (наприклад, значення *-3*). На другому перегляді ця директива не обробляється.

При лексичному аналізі необхідно відслідковувати появу визначеного ідентифікатора з типом *-3* і замінювати цей ідентифікатор на послідовність символів із таблиці ідентифікаторів користувача з продовженням аналізу.

Обробка директиви =

Обробка здійснюється за алгоритмом 2 (для обох переглядів):

1) Обчислити абсолютний вираз поля операндів директиви згідно з алгоритмом Дейкстри (див. стор. 25 ***Обробка абсолютних виразів***).
2) Провести пошук ідентифікатора з поля імені директиви в таблиці ідентифікаторів користувача. При успіху пошуку – перейти до п.4.

3) Записати новий ідентифікатор в таблицю, задавши значення зміщення як значення абсолютного виразу поля операндів та значення типу – *abs* (наприклад, значення *0*). Перейти до п.5.
4) Записати нове значення зміщення у рядок таблиці знайденого ідентифікатора.
5) Кінець.

Обробка директиви Label

Виконується Алгоритм 1, в якому значення типу визначається за ключовим словом у полі операндів директиви.

Визначення кількості байтів, які генеруються за рядками програми

Після обробки поля імені (мітки) визначається кількість байтів, які Асемблер повинен генерувати за поточним реченням. Детальний алгоритм визначення кількості байтів розглянемо далі. Тут відмітимо, що визначене значення *k* кількості байтів додається до значення поля "*Поточне зміщення*" активного сегмента у таблиці сегментів (тобто *$ = $+ k*).

Розглянутий процес формування таблиці ідентифікаторів, визначених користувачем, та визначення кількості байтів за реченням називають *процесом автоматичного розподілу пам'яті програми* або просто *розподілом пам'яті*.

Визначення кількості байтів даних у директивах db, dw, dd, dp, dq, dt

На *обох переглядах* значення кількості байтів *k*, які необхідно згенерувати за відповідною директивою, визначається за формулою

$$k = type*count$$

де *type* (тип) визначається директивою, а *count* (у випадку відсутності операндів з повтореннями) – це кількість операндів директиви згідно з таблицею структури речення (табл. 4.4).

У випадку застосування операнда з оператором повторення *dup*, *count* = кількість повторень. Якщо ж у директиві використовуються операнди з вкладеними повтореннями або операнди як з повтореннями, так і без них, тоді для визначення *count* рекомендується використовувати рекурсивну процедуру.

Якщо у директиві *db* використовується текстова константа, то *count* – це кількість символів відповідної лексеми з таблиці лексем.

На *другому перегляді* додатково необхідно згенерувати байти даних, попередньо обчисливши значення констант чи абсолютних виразів. При генеруванні байтів необхідно пам’ятати, що в процесорах фірми Intel молодші байти розміщуються за молодшими адресами. Деякі особливості має генерування, якщо в директивах *dw, dd, dp* в якості операнда задається мітка або символічне ім’я. Процес залежить від розрядності, яка задана в поточному сегменті (де знаходяться директиви *dw, dd, dp*) та розрядності сегмента, де знаходиться ім’я (мітка), і, відповідно, розрядності зміщення імені (мітки).

Правила відповідають здоровому глузду. В директиві *dw* може використовуватись лише 16-розрядне зміщення у 16-розрядному сегменті, при цьому із таблиці ідентифікаторів користувача читається відповідне 16-розрядне зміщення і на його основі генеруються байти програми. У директиві *dd* в аналогічному випадку додатково генеруються ще два старших нульових байти для значення сегментної складової, яку визначить завантажувач. У директиві *dd*, у випадку 32-розрядного зміщення та 32 розрядного поточного сегмента, генеруються 4 байти цього зміщення. У

такому ж випадку, для директиви *dp* додатково генеруються ще два старших нульових байти для значення сегментної складової. Всі інші випадки помилкові (хоча *Tasm* не завжди вказує на помилку).

Визначення кількості байтів машинної інструкції (на обох переглядах) та генерування команд процесора на другому перегляді

На обох переглядах обробка адресних виразів використовується для визначення кількості байтів, які необхідно згенерувати за поточною машинною інструкцією.

Нагадаємо структуру коду команди процесорів 80x86 (рис. 4.3):

1, 2 або 3 однобайтні префікси	1 або 2 байти коду операції	0, 1 або 2 байти режиму адресації (ModR/m та Sib)	0, 1, 2 або 4 байти зміщення в команді	0, 1, 2 або 4 байти безпосередніх даних

Рисунок 4.3 – Структура коду команди процесорів 80x86

Розглянемо окремо кожне поле коду команди.

Код операції

Кількість байтів коду операції однозначно визначається мнемонікою команди і записується в один із стовпців таблиці ключових слів. У деяких командах частина коду операції розташовується в полі *reg* байта *modR/m*. У цьому випадку, у відповідний рядок таблиці ключових слів записується, що код операції є однобайтним, та ознака безумовної присутності байта *modR/m*. В результаті, значення кількості байтів з таблиці ключових слів заноситься як початкове значення кількості байтів *k*, які формуються на основі поточної інструкції.

В той же час, значення байту (байтів) коду операції (на другому перегляді) далеко не завжди визначається мнемонікою. Необхідний додатковий аналіз поля операндів. При цьому існують мнемоніки, коли коди операції кардинально залежать від поля операндів. Це дещо ускладнює структуру рядків таблиці ключових слів, які презентують мнемоніки машинних інструкцій, оскільки необхідно заносити в рядок усі можливі заготовки для кодів операцій. Наприклад, значна кількість команд в 0-му біті коду операції містить *ознаку розрядності даних 0* – для байтів, *1* – для слів або подвійних слів (конкретне значення – за замовчуванням або змінюється при наявності префікса заміни розрядності даних – див. наступні пункти). Крім того, для багатьох команд *1-й* біт коду операції вказує, який саме операнд є приймачем: *0* – приймач визначається полем *r/m, 1* - полем *reg*.

Визначення розрядності зміщення в команді, наявності префікса зміни розрядності адрес

Поле зміщення присутнє в команді, якщо один з операндів є адресним виразом, який, у свою чергу, містить визначений або невизначений ідентифікатор користувача, абсолютний вираз чи константу. У випадку наявності регістрів адрес (регістри у квадратних дужках), визначається можлива розрядність поля зміщення в команді – *1* або *2* байти при 16-розрядних адресних регістрах, *1* або *4* байти при 32-розрядних.

Якщо адресний вираз містить адресний терм або невизначений ідентифікатор, кількість байтів зміщення в команді може дорівнювати *2* або *4*, навіть у випадку, коли зміщення в сегменті для такого ідентифікатора є однобайтним. Це пояснюється тим, що однобайтне зміщення в сегменті під час редагування зв'язків може стати двох- або

чотирьохбайтним. Але при роботі редактора зв'язків вставити у програму додатковий байт практично неможливо.

Якщо ж в адресному виразі немає адресного терму або невизначеного ідентифікатора, розрядність поля зміщення в команді визначається значенням абсолютного виразу чи константи: *1* байт (якщо значення більше або дорівнює -128 і менше 128), *2* або *4* байти (в інших випадках).

Визначена кількість байтів зміщення в команді (*0, 1, 2* або *4*) додається до *k*.

На *другому перегляді* до зміщення адресного терму (якщо він є) додається значення абсолютного виразу (якщо він є) та за отриманим результатом генеруються відповідні байти поля зміщення в команді.

Якщо розрядність регістрів (регістра) адрес відрізняється від значення в полі розрядності активного сегмента, тоді встановлюється $k = k+1$, а на *другому перегляді* додатково генерується *префікс зміни розрядності адрес* (код – 67h).

Розрядність даних, наявність префіксів зміни розрядності даних

Розрядність даних у машинній інструкції визначається розрядністю регістра даних або оператором *ptr*, або типом адресного терму.

Якщо в операндах машинної інструкції існує більш ніж один із перелічених варіантів, то всі вони повинні вказувати на одну й ту саму розрядність.

Якщо в операндах машинної інструкції немає жодного з перелічених варіантів, це свідчить про помилку в інструкції.

Якщо визначена розрядність 8, префікс зміни розрядності даних не генерується.

Якщо визначена розрядність даних 16 або 32, а в полі розрядності активного сегмента вказана інша розрядність, тоді установлюється $k = k+1$,

а на другому перегляді додатково генерується *префікс зміни розрядності даних* (код – 66h).

Якщо в машинній інструкції є операнд, який є абсолютним виразом або константою, кількість байтів безпосередніх даних відповідає визначеній вище розрядності даних. У ряді команд допускається використання однобайтних безпосередніх даних при розрядності даних у два або 4 байти та значенні абсолютного виразу в межах від -128 до 127. При цьому відповідний біт коду операції встановлюється в 1, а при виконанні команди процесор виконує знакове розширення однобайтних безпосередніх даних.

Байт префіксу заміни сегмента

Алгоритм визначення байта неявно заданого префікса заміни сегмента полягає у наступному:

1) Визначається сегментний регістр за замовчуванням (за переліком регістрів адрес). Нагадаємо, що, як правило, за замовчуванням використовується регістр *DS*. Виключення наступні: у випадку використання *BP* як адресного регістра або регістрів *EBP* і *ESP* (без множника) як базових, за замовчуванням використовується регістр *SS*.
2) Якщо операнди машинної інструкції не є адресними виразами, або в адресних виразах відсутній адресний терм, або це команда передачі управління з типом операнда *Far* (*Near*), байт заміни сегмента відсутній.
3) За таблицею ідентифікаторів користувача визначається ім'я логічного сегмента, в якому розміщений адресний терм.

4) У таблиці назначень сегментним регістрам (див. стор. 31 ***Обробка директиви Assume***) визначається сегментний регістр, якому назначено логічний сегмент, де саме розміщується ідентифікатор.
5) Якщо логічному сегменту назначено сегментний регістр, який не співпадає з сегментним регістром за замовчуванням, тоді повинен генеруватись байт префікса заміни сегмента назначеного сегментного регістра та встановлюватись $k = k+1$ (значення байту заміни сегмента для генерації на *другому перегляді* згідно з Додатком Б).

При явному завданні префікса заміни сегмента визначається сегментний регістр за замовчуванням (згідно з вищезазначеним), і, при його співпадінні з явно заданим сегментним регістром, префікс не генерується.

Байти режимів адресації

На обох переглядах необхідно визначити наявність та кількість байтів (один чи два) режимів адресації. На *другому перегляді* додатково визначаються значення цих байтів.

Можна виділить наступні групи машинних інструкцій:

- перша група – поле режиму адресації для заданої мнемоніки завжди відсутнє;
- друга група – поле режиму адресації для заданої мнемоніки завжди присутнє;
- третя група – поле режиму адресації може бути і залежить від поля операндів;
- четверта група – одній і тій самій машинній інструкції мовою Асемблера можуть відповідати дві машинні команди з

абсолютно однаковими діями, але з байтами режимів адресації або без них.

До першої групи відносяться, наприклад, команди управління ознаками (*Cli, Sti, Clc, Stc, Cmc, Cld, Std*), команди корекції для двійково-десяткових обчислень (*Aaa, Aas, Daa, Das*), команди передачі управління за умовою, команди обробки масивів – рядкові (ланцюгові) команди, команди перетворення типів (байт-слово, слово-подвійне слово) та ін.

До другої групи належать, наприклад, команди *Lea, Movsx, Movzx, Neg, Not*, команди обробки бітових полів, команди множення та ділення беззнакових чисел, команди завантаження логічної адреси (*Lds, Lss, Les, Lgs, Lfs*), команди лінійних, арифметичних та циклічних зсувів.

До третьої групи відносяться команди *Jmp* та *Call*. При використанні прямої (відносної) адресації поле режиму адресації відсутнє. Якщо використовується опосередкована (непряма) адресація, тоді присутнє поле режиму адресації.

До четвертої групи відносяться команди традиційної обробки: пересилання, додавання, віднімання, порівняння, порозрядної логічної обробки. Оскільки ці команди найчастіше застосовуються в програмах, для частини з них, поряд із загальними, реалізовані спрощені формати структури команди без поля режимів адресації. Наприклад, для команд, де першим з операндів є регістр *EAX* (*AX, AL*), а другим – безпосередні дані та ін. Це потребує більш детального аналізу команд вказаної групи для визначення наявності поля режиму адресації.

Поле режиму адресації складається із байта *Modr/m* та, можливо, байта *sib*. Наявність байту *sib* визначається адресним виразом при виконанні однієї із умов: наявності двох 32-розрядних адресних регістрів або наявності множника.

Формування байтів поля режимів адресації на другому перегляді

Структура байта *Modr/m* при 16-розрядній адресації має наступний вигляд (рис. 4.4):

Поле mod – 2 біти	Поле reg – 3 біти	Поле r/m – 3 біти

Рисунок 4.4 – Структура байта Modr/m при 16-розрядній адресації

Поле *mod* використовується для визначення зміщення в команді, а поле *r/m* – для визначення регістрів адрес, вміст яких використовується для формування ефективної адреси. Поле *reg* призначене для завдання регістра даних, який використовується в команді або є частиною коду операції.

Розглянемо поле mod:

- При *mod=00* зміщення в команді відсутнє, а ефективна адреса формується за вмістом регістра (регістрів) адрес, які задаються полем *r/m*.
- При *mod=01* зміщення в команді однобайтне, яке при формуванні ефективної адреси знаково розширюється до двох байтів з подальшим додаванням до вмісту регістра (регістрів) адрес, які задаються полем *r/m*.
- При *mod=10* зміщення в команді двохбайтне і при формуванні ефективної адреси додається до вмісту регістра (регістрів) адрес, які задаються полем *r/m*.
- При *mod=11* адреса пам'яті не задається, а поле *r/m* задає код регістра даних.

У полі *r/m* регістри адрес або їх можлива комбінація задається наступним чином (табл. 4.10):

Таблиця 4.10 – Поле *r/m* у режимі 16-розрядної адресації

Код у полі r/m	Регістри адрес	Сегментний регістр, який використовується за замовчуванням
000	BX+SI	DS
001	BX+DI	DS
010	BP+SI	SS
011	BP+DI	SS
100	SI	DS
101	DI	DS
110	BP	SS
111	BX	DS

З табл. 4.10 можна зробити наступні висновки, які справедливі також і для сучасних мікропроцесорів сімейства при 16-розрядній адресації:

1. При 16-розрядній адресації тільки чотири регістра – *BX, BP, SI* та *DI* – можуть використовуватися як регістри адрес.
2. Для формування багатокомпонентної адреси використовується тільки обмежений набір пар регістрів: (*BX, SI*), (*BX, DI*), (*BP, SI*), (*BP, DI*).
3. Із реалізації випадає один із широко вживаних режимів – режим прямої адресації, тобто режим, коли зміщення в команді і є зміщенням у сегменті.

Відносно останнього пункту інженери фірми Intel вимушені були прийняти наступне вирішення: ввести режим прямої адресації при *mod=00* та *r/m=110*, тобто, не дивлячись на *mod=00*, зміщення в команді задавати двохбайтним, якщо *r/m=110*. При цьому регістр *BP* не використовується. Це дуже нагадує "латку" в програмах. Але ця "латка" досить продумана. З апаратної реалізації випадає режим посередньої регістрової адресації з використанням регістра *BP* як регістра адреси, проте використання цього режиму при стратегічному призначенні регістра *BP* як базового регістра

структур даних стеку малоймовірне. У крайньому випадку, можна використати режим при *mod=01* та нульовому байті зміщення в команді, що і реалізовано компіляторами програм мови Асемблера.

Використання засобів формування 32-розрядних адрес у реальному режимі

Свого часу при створенні процесора 80386 перед інженерами фірми Intel постала непроста проблема: з одного боку, необхідно переходити на 32-розрядні дані та адреси, щоб забезпечити конкурентоздатність нового процесора, а з іншого боку, необхідно забезпечити програмну сумісність нового процесора зі старими. Очевидне вирішення проблеми – створення додаткових кодів операцій – було б неоптимальним. Дійсно, при цьому кількість кодів операцій необхідно було б збільшити у декілька разів (поряд з існуючими командами обробки 16- та 8-розрядних даних при 16-розрядних адресах, необхідно було б створити команди обробки 8, 16 та 32-розрядних даних при 32-розрядних адресах та команди обробки 32-розрядних даних при 16-розрядних адресах). Ця проблема була вирішена наступним чином.

По перше, встановлювалось глобальне (для програми в цілому) значення розрядності адрес та даних за замовчуванням: або 16, або 32. Наприклад, у реальному режимі, за замовчуванням, для всіх програм встановлювались 16-розрядні адреси та 8- або 16-розрядні дані, що властиве старим процесорам.

По друге, створювались однобайтні префікси команд для зміни розрядності, яка задана за замовчуванням (*66h – зміна глобального значення розрядності даних*, *67h – зміна глобального значення розрядності команд*). Дія цих префіксів обмежувалась рамками однієї команди, яка розташовувалась за префіксами. У реальному режимі наявність префіксу

66h забезпечувала при одному й тому ж коді операції використання 32-розрядних даних замість 16-розрядних. Якщо ж код операції вказував на 8-розрядні дані, тоді цей префікс на таку команду не впливав (та і необхідність префіксу *66h* перед командами обробки 8-розрядних даних відпадає). Наявність префіксу *67h* дозволяє використовувати більш гнучкий механізм 32-розрядної адресації.

Таким чином, у реальному режимі забезпечувалось виконання усіх програм, розроблених для попередніх процесорів. Щодо нових програм, то вони могли обробляти 32-розрядні дані та використовувати механізм формування 32-розрядних адрес без створення додаткових кодів операцій.

В той же час, при модифікації мови Асемблера засоби явного завдання префіксів зміни розрядності не вводились. Формування префіксів зміни розрядності повністю покладалось на компілятор.

Інтерпретація полів байта *mod-r/m* у режимі 32-розрядної адресації відрізняється від режиму 16-розрядної адресації. Головна відмінність – в інтерпретації процесором поля *r/m* (табл. 4.11):

Таблиця 4.11 – Поле *r/m* у режимі 32-розрядної адресації

Код у полі r/m	Регістри адрес	Сегментний регістр, який використовується за замовчуванням
000	EAX	DS
001	ECX	DS
010	EDX	DS
011	EBX	DS
100	Наявність байта *Sib*	--
101	EBP*	SS
110	ESI	DS
111	EDI	DS

Байт *SIB* має наступну структуру (рис. 4.5):

Поле множника – 2 біти	Поле індексного регістра - 3 біти	Поле базового регістра - 3 біти

Рисунок 4.5 – Структура байта *SIB*

У полі індексного регістра може вказуватись будь-який регістр за виключенням *ESP*. При формуванні ефективної адреси вміст регістра зсувається вліво на кількість розрядів, які вказані у полі множника. Тим самим фактично при формуванні ефективної адреси відбувається множення вмісту індексного регістру на 2, 4 або 8, що позбавляє програміста додаткових дій при адресації елементів масивів слів, подвійних слів та квадрослів.

У полі базового регістра може використовуватися будь-який регістр загального призначення (*РЗП*), у тому числі і *ESP*.

Таким чином, можливість формування ефективної адреси у 32-розрядному режимі значно ширша, порівняно з 16-розрядним режимом адрес. Тому 32-розрядний режим за допомогою префікса зміни розрядності адрес може використовуватися і в реальному режимі при наступному уточненні – старші 16 розрядів ефективної адреси у реальному режимі ігноруються.

Окрім іншої інтерпретації поля *r/m*, по-іншому інтерпретується поле *mod*:

- при *mod=01* зміщення в команді однобайтне, яке при формуванні ефективної адреси знаково розширюється до

чотирьох байтів з наступним додаванням до вмісту регістра (регістрів) адрес, які задаються полем *r/m* та *sib*;

- при *mod=10* зміщення в команді чотирьохбайтне і при формуванні ефективної адреси додається до вмісту регістра (регістрів) адрес, які задаються полем *r/m* та *sib*.

Приклади адресації у 32-розрядному режимі:

```
Mov  ax,Value[ecx*4+edx]
Mov  edx,[edx*8]
Mov  al,[eax+esp]
```

Як і у випадку 16-розрядних адрес, режим використання регістра *EBP* при *mod=0* відсутній. При коді *101* у полі *r/m* або у полі базового регістра байта *sib* встановлюється режим використання 4-х байтного зміщення у команді.

Особливості трансляції окремих команд

Команди передачі управління

Команди передачі управління, з точки зору мови Асемблера, поділяються на:

- команди з прямою адресацією (прямі), коли в операнді команди вказується мітка або ім’я процедури;
- опосередковані (непрямі), коли в операнді команди вказується адреса даних, які містять адресу переходу.

Трансляція непрямих команд передачі управління мало чим відрізняється від раніше викладеного, тому їх окремо розглядати немає сенсу.

Команди передачі управління поділяють також на внутрішньо-сегментні та міжсегментні. Внутрішньо-сегментні прямі команди передачі управління з точки зору процесора мають відносну адресацію, коли

цільова адреса формується як алгебраїчна сума вмісту (E)IP та зміщення в команді.

Команди передачі управління за умовою

Команди передачі управління за умовою можуть мати одну з наступних структур (рис. 4.6).

Визначається значення зміщення адресного терму в полі операндів.

Один байт коду операції	Один байт зміщення у команді
Два байти коду операції	Два (чотири для 32 розрядних адрес) байти зміщення у команді

Рисунок 4.6 – Структура команди передачі управління за умовою

На *першому перегляді* алгоритм обробки команди передачі управління за умовою є наступним:

1) Визначається наявність мітки в полі операндів. Якщо в таблиці ідентифікаторів користувача відповідна мітка відсутня, тоді перейти до п.5).
2) Читається значення мітки з таблиці, і, якщо операнд додатково має абсолютний вираз, він обчислюється, і отримане значення додається до значення зміщення мітки.
3) Обчислюється різниця між отриманим значенням та значенням поля «*Поточне зміщення*» відкритого логічного сегмента, збільшеним на довжину команди (*$+2*).
4) Якщо різниця менша за -128 або більша за 127, потрібно перейти до п.65).
5) Прийняти *k=2*. Перейти до п.7).

6) Прийняти *k=4*. (*k=6* при 32-розрядній адресації)

7) Кінець.

На *другому перегляді* всі мітки визначені, тому для визначення посилання вперед використовується порівняння зміщення цільової мітки (*Зм*) плюс значення абсолютного виразу (*Абс*), якщо він є, та значення *$*. Якщо (*Зм+Абс*) - *$ > 2*, то безумовно генерується 4-х байтна або 6-ти байтна команда, інакше – 2-х байтна.

Для генерації чотирьохбайтного зміщення команди передачі управління використовується значення виразу *Зм-$-4*, а 2-х байтного – значення виразу *+Зм-$-2*.

ЗАУВАЖЕННЯ 1. При посиланні вперед для команд передачі управління за умовою на *першому перегляді* відводяться 4 байти. Якщо на *другому перегляді* виявиться, що можна згенерувати 2-х байтну команду, то «лишні» два байти заповнюються командою *NOP* (код операції *90h*). Так працює компілятор TASM. (В компіляторі MASM така можливість не використовується, а генерується 4-байтна команда).

ЗАУВАЖЕННЯ 2. Якщо в команді мовою Ассемблера задано оператор *Short*, то *на першому перегляді* безумовно встановлюється *k=2*. Якщо на *другому перегляді* виявиться, що відстань більша за 127, то генерується повідомлення про помилку.

ЗАУВАЖЕННЯ 3. Якщо ім’я поточного сегменту кодів відрізняється від імені сегмента кодів, де розташована мітка, то генерується повідомлення про помилку.

Внутрішньосегментні команди безумовної передачі управління з прямою (для Асемблера і відносною для процесора) адресацією

Ці команди мають один байт коду операції і один або два (чотири для 32-розрядної адресації) байти зміщення в команді. Крім цього, їх трансляція аналогічна трансляції команд передачі управління за умовою.

5. ЕТАПИ ВИКОНАННЯ І ПОРЯДОК ЗАХИСТУ КУРСОВОЇ РОБОТИ

5.1. Графік виконання курсової роботи

Для ефективного планування часу виконання курсової роботи студенту надається календарний план-графік, який містить основні етапи КР та терміни їх виконання (табл. 5.1).

Таблиця 5.1 – Календарний план-графік виконання КР

№ пп	Етапи виконання	Термін виконання (тижні семестру)	Форма звітності
1	2	3	4
1.	Отримання студентом індивідуального завдання на виконання курсової роботи	1	Фіксується дата отримання завдання
2.	Створення на базі варіанту завдання тестових програм мовою Асемблера. Узгодження програм з викладачем	3	Роздруківки файлів: test1.asm (для створюваного компілятора), test2.asm + test2.lst (для TASM.exe)
3.	Розробка лексичного аналізатора, формування таблиці лексем та таблиці структури речення	5	Працююча програма з роздруківкою результатів – створеного файлу з таблицями лексем та структурою речення *

Продовження табл. 5.1

1	2	3	4
4.	Розробка програми 1-го перегляду	7	Працююча програма з роздруківкою результатів – лістингу 1-го перегляду *
5.	Розробка програми 2-го перегляду	9	Працююча програма з роздруківкою результатів– лістингу 2-го перегляду *
6.	Оформлення результатів курсової роботи	10	Пояснювальна записка + папка з модулями програми (щодо вмісту див. п.5.2)
7.	Захист курсової роботи	11	Працююча програма + оформлені результати

* – **Під час контролю виконання окремих етапів і захисту КР керівник може запропонувати внести у тестовий файл будь-які зміни.**

Тестові програми та результати роботи розроблених модулів компілятора для 3-го, 4-го та 5-го етапів, що відповідають створеній на етапі 2 тестовій програмі, мають бути представлені керівнику у роздрукованому вигляді та у встановлені планом-графіком терміни для поточного контролю роботи студента.

Керівник здійснює контроль за ходом роботи, надає студенту необхідну консультативну допомогу при вивченні теоретичного матеріалу та розробленні програмного забезпечення, дає висновок про допуск роботи до захисту.

5.2. Вимоги до оформлення результатів курсової роботи

За результатами виконання курсової роботи і для допуску до захисту КР студент повинен підготувати:

1. Роздруковану та підписану автором **Пояснювальну записку.**
2. Підготовлений каталог з наступним ім’ям – останні дві цифри поточного року і прізвище студента. Наприклад, **20Петренко**. При наявності студентів з однаковим прізвищем додається ім’я і т.д. В каталог записуються всі початкові файли розробленого компілятора. Розширення імен файлів, незалежно від мови програмування, потрібно замінити на .txt, наприклад - lexemes.txt.
3. Виконавчий файл компілятора.
4. Файли тестових програм.

Пояснювальна записка повинна містити:

1. Титульну сторінку з загальноприйнятим вмістом (зразок у Додатку А).
2. Загальне завдання та індивідуальне завдання (за варіантом).
3. Опис загальної структури розробленої програми, окремих модулів і підпрограм та їх взаємодії.
4. Роздруківки результатів кожного з етапів курсової роботи, підписані викладачем.

5.3. Процедура захисту курсової роботи

До захисту курсової роботи допускаються студенти, які виконали її в повному обсязі та у встановлені терміни і представили відповідні матеріали керівнику.

У разі недопуску курсової роботи до захисту студент повинен врахувати зауваження керівника курсової роботи та доопрацювати її з повторною подачею керівникові у встановлений термін.

Захист курсової роботи проходить за встановленим графіком захисту.

За тиждень до захисту студент повинен представити керівнику:

- оформлену пояснювальну записку;
- розроблене програмне забезпечення (див. п. 5.2).

Під час захисту студент має продемонструвати працездатність та коректність роботи створеного ним програмного продукту на комп'ютері, використовуючи підготовлені ним тестові файли програм мовою Асемблера, що засвідчить факт виконання студентом поставлених завдань.

Студенту може бути запропоновано внести незначні зміни у завдання та модифікувати відповідно до них програму. При виявленні під час демонстрації незначних помилок можливе доопрацювання програми студентом на місці.

Після демонстрації роботи студенту можуть бути поставлені запитання з тематики, вмісту та результатів його роботи, на які він повинен дати чітку й обґрунтовану відповідь.

При визначенні оцінки за курсову роботу враховується поточна робота студента над завданням на КР, результати роботи та її захисту, а саме:

- якість виконання окремих етапів КР, дотримання студентом затвердженого календарного плану виконання курсової роботи;
- якість та своєчасність виконання курсової роботи в цілому;
- відповідність оформлення пояснювальної записки нормативним вимогам та технічному завданню;
- самостійність розробки;

- правильність і аргументованість відповідей на запитання.

За дострокове та якісне виконання окремих етапів виконання курсової роботи та роботи в цілому студенту можуть бути нараховані **заохочувальні бали.**

СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ

1. Зубков С.В. Assembler. Язык неограниченных возможностей. Приложение 2. Команды Intel 80x86. [Текст] / С.В. Зубков – М., 2001.
2. Юров В. Assembler. Учебный курс. Урок 5. Синтаксис ассемблера. [Текст] / В. Юров – СПб. : Питер, 2001.
3. Квиттнер П. Задачи Программы Вычисления Результаты. Гл 5. Языки ассемблеров и ассемблеры. [Текст] / П. Квиттнер – М. Мир, 1980.
4. Лебедев В.Н. Введение в системы программирования. (в части организации вычисления выражений). [Текст] / В.Н. Лебедев – М. Статистика, 1975.

ДОДАТКИ

Додаток А Зразок оформлення титульного аркуша курсової роботи

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
Кафедра системного програмування і спеціалізованих комп'ютерних систем

КУРСОВА РОБОТА

з дисципліни «Системне програмування»
на тему: Розробка компілятора програм мовою Асемблера

Студента (ки) ____ курсу _____ групи
за спеціальностю
123 «Комп'ютерна інженерія»
_____
(прізвище та ініціали)
Керівник_____
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Національна оцінка _____
Кількість балів: _____Оцінка: ECTS ____

Члени комісії _____ _____
(підпис) (вчене звання, науковий ступінь, прізвище та ініціали)
_____ _____
(підпис) (вчене звання, науковий ступінь, прізвище та ініціали)
_____ _____
(підпис) (вчене звання, науковий ступінь, прізвище та ініціали

Київ- 2021 рік

Додаток Б Основні регістри процесора

8-розрядні регістри

Ім'я регістра	Номер[1]	Характеристика
AL	000	**Регістр даних**. Молодша частина AX. При використанні цього регістра команди обробки безпосередніх даних та команди пересилання більш компактні та їх виконання потребує менше часу.
CL	001	**Регістр даних**. Молодша частина CX. Додатково використовується у командах зсуву для вказання кількості зсувів.
DL	010	**Регістр даних**. Молодша частина DX.
BL	011	**Регістр даних**.. Старша частина BX.
AH	100	**Регістр даних**.. Старша частина AX. Додатково використовується як розширення регістра AL у командах множення та ділення 8-розрядних даних
CH	101	**Регістр даних**. Старша частина CX.
DH	110	**Регістр даних**. Старша частина DX.
BH	111	**Регістр даних**. Старша частина BX.

[1] Вказується двійковий код регістра, що використовується в кодах команд

16-розрядні регістри

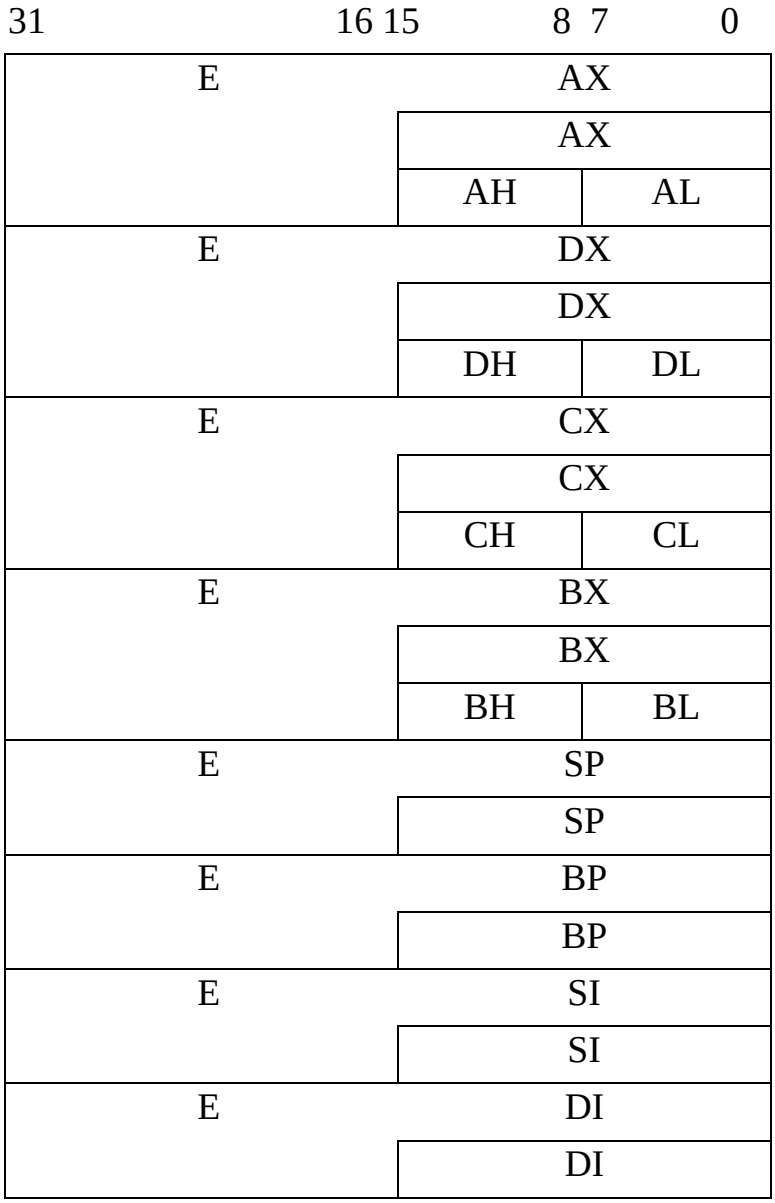
Ім'я регістра	Номер	Характеристика	Сегментний регістр[2]
AX	000	**Регістр даних**. Молодша частина EAX. При використанні цього регістру команди обробки безпосередніх даних та команди пересилання більш компактні та їх виконання потребує менше часу.	-
CX	001	**Регістр даних**. Молодша частина ECX. Додатково використовується як лічильник циклів у рядкових (ланцюгових) командах і командах організації циклів.	-
DX	010	**Регістр даних**. Молодша частина EDX. Додатково використовується як розширення регістру AX у командах множення і ділення 16-розрядних даних. Використовується для адресації портів введення-виведення у командах введення-виведення.	-
BX	011	**Регістр загального призначення**. Молодша частина EBX. Типове використання – для адресації складних структур даних.	DS
SP	100	**Вказівник стеку**. Молодша частина ESP. В окремих випадках може використовуватись як **регістр даних**.	SS
BP	101	**Регістр загального призначення**. Молодша частина EBP. Типове використання – для адресації складних структур даних у стеку.	SS
SI	110	**Регістр загального призначення**. Молодша частина ESI. **Особливе** використання – адресація елементів джерела у рядкових (ланцюгових) командах. Типове використання – адресація елементів масивів.	DS
DI	111	**Регістр загального призначення**. Молодша частина EDI. **Особливе** використання – для адресації елементів рядка приймача у рядкових (ланцюгових) командах. Типове використання – адресація елементів масивів.	DS (ES -у ланцюгових командах)

[2]Вказується сегментний регістр, який використовується за замовчуванням за умови, що відповідний регістр загального призначення використовується як адресний.

32-розрядні регістри

Ім'я регістра	Номер	Характеристика	Сегмент-ний регістр
EAX	000	**Регістр загального призначення.** При використанні цього регістра команди обробки безпосередніх даних і команди пересилання більш компактні та їх виконання потребує менше часу.	DS
ECX	001	**Регістр загального призначення**. Додатково використовується як лічильник циклів у рядкових командах і командах організації циклів	DS
EDX	010	**Регістр загального призначення**. Додатково використовується як розширення регістру EAX у командах множення і ділення 32-розрядних даних.	DS
EBX	011	**Регістр загального призначення.** Типове використання – для адресації складних структур даних.	DS
ESP	100	**Вказівник стеку**. В окремих випадках може використовуватися як **регістр даних**.	SS
EBP	101	**Регістр загального призначення**. Типове використання – для адресації складних структур даних у стеку.	SS
ESI	110	**Регістр загального призначення**. Типове використання – адресація елементів масивів.	DS
EDI	111	**Регістр загального призначення.** Типове використання – адресація елементів масивів	DS (ES – у ланцю-гових командах)

Регістрова модель



Сегментні регістри

Складаються з явної 16-розрядної частини та 64-розрядної тіньової. В явну записуються старші 16 біт фізичної адреси сегмента (реальний режим), або селектор дескриптора сегмента (захищений режим). У захищеному режимі, одночасно з завантаженням селектора дескриптора в явну частину, сам дескриптор із пам’яті завантажується у тіньову частину.

Ім'я регістра	Номер	Характеристика	Префікс
ES	000	Містить старші 16 біт фізичної адреси допоміжного сегмента даних (реальний режим) або селектор дескриптора допоміжного сегмента даних (захищений режим). **За замовчуванням, використовується для адресації операнда приймача для рядкових (ланцюгових) команд** без можливості заміни. При наявності перед командою префікса 26h (ES:*операнд_пам'яті* мовою Асемблера) використовується замість сегментних регістрів DS та SS, заданих за замовчуванням.	26h (ES:)
CS	001	Містить старші 16 біт фізичної адреси сегмента кодів (реальний режим) або селектор дескриптора сегмента кодів (захищений режим). За замовчуванням використовується для адресації команд без можливості заміни. При наявності перед командою префікса 2Eh (CS:*операнд_пам'яті* мовою Асемблера) використовується замість сегментних регістрів DS та SS, заданих за замовчуванням.	2Eh (CS:)
SS	010	Містить старші 16 біт фізичної адреси сегмента стеку (реальний режим) або селектор дескриптора сегмента стеку (захищений режим). **За замовчуванням,** використовується для адресації верхівки стеку у командах **push, pop, call, ret** без можливості заміни. За замовчуванням, з можливістю заміни застосовується в усіх інших командах звернення до пам'яті, де для формування ефективної адреси використовується вміст регістрів **BP, EBP або ESP**. При наявності перед командою префікса 36h (SS:*операнд_пам'яті* мовою Асемблера) використовується замість сегментного регістра DS, заданого за замовчуванням.	36h (SS:)
DS	011	Містить старші 16 біт фізичної адреси сегмента даних (реальний режим) або селектор дескриптора сегмента даних (захищений режим). За замовчуванням, з можливістю заміни, використовується для адресації даних у всіх випадках, коли за замовчуванням не використовуються інші сегментні регістри. При наявності перед командою префікса 3Eh (DS:*операнд_пам'яті* мовою Асемблера) використовується замість сегментного регістра SS, заданого за замовчуванням.	3Eh (DS:)
FS	100	Містить старші 16 біт фізичної адреси сегменту даних (реальний режим), або селектор дескриптора сегменту даних (захищений режим). **За замовчуванням не використовується**. При наявності перед командою префікса 64h (FS:*операнд_пам'яті* мовою Асемблера) використовується замість сегментних регістрів DS та SS, заданих за замовчуванням.	64h (FS:)
GS	101	Містить старші 16 біт фізичної адреси сегмента даних (реальний режим) або селектор дескриптора сегмента даних (захищений режим). **За замовчуванням не використовується**. При наявності перед командою префікса 65h (GS:*операнд_пам'яті* мовою Асемблера) використовується замість сегментних регістрів DS та SS, заданих за замовчуванням.	65h (GS:)

Регістр ознак (Flags)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
			of	df		if	tf	sf	zf	0	af	0	pf	1	cf

cf – ознака перенесення;

pf – ознака парної кількості 1;

af – ознака допоміжного перенесення;

zf – ознака *0* (*0* – число **не дорівнює** *0*, *1* – число **дорівнює** *0*);

sf – ознака знаку числа (*0* – додатне, *1* – від'ємне);

df – ознака напрямку;

of – ознака переповнення.

Додаток В Тестовий файл для прикладу індивідуального завдання

```
Data1       Segment
Vb          db          10011b
String      db          ‘Рядок – new’
Val         db          -128
Vw          dw          4567d
Vd          dd          0d7856fdah
Data1       ends
Data2       Segment
Doublesg    dw          678
QWERTY      dd          67ff89h
Zxcv        db          89h
Data2       Ends
Assume cs:Code,Ds:Data1,Gs:Data2
Code        Segment
            Jmp         label1
label1:
            Cli
            Inc         cl
            Jb          Label2
            Inc         Bx
            Dec         Vw[si]
            Dec         gs:zxCV[bp]
            Add         Eax, Esi
            Cmp         Ax, Doublesg[edi]
            Cmp         ebx, qwerty[ebx]
            Xor         vb[edx], al
            Mov         dx, 5634h
            Or          Vd[esp], 0101b
            Jb          label1
Label2:
            Jmp         Label2
Code        ends
```

Додаток Г Приклад трансляції макросів

Розглянемо особливості виконання другого етапу КР при наявності макросів.

Обробка директиви Macro

Спочатку директива обробляється на загальних підставах, тобто створюється таблиця лексем і таблиця структури речення, які виводяться на друк. Наприклад, задано наступне макровизначення

```
Testm2 MACRO T1
        Dec T1
endm
```

Дії на першому перегляді полягають у наступному. Створюємо відповідні таблиці (табл. Г.1-Г.2):

Таблиця Г.1. Таблиця лексем для TESTM2 MACRO T1

№пп	Лексема	Довжина лексеми у символах	Тип лексеми
1	TESTM2	6	IDENTIFIER
2	MACRO	5	DIRECTIVE
3	T1	2	IDENTIFIER

Таблиця Г.2. Таблиця структури речення для TESTM2 MACRO T1

Поле міток	Поле мнемокоду		Поле 1-го операнда	
№ лексеми поля	№ першої лексеми поля	Кількість лексем поля	№ першої лексеми операнда	Кількість лексем операнда
1	2	1	3	1

Ім’я макроса (TESTM2) **записується в таблицю ключових** слів компілятора з ознакою макрокоманди і з вказівкою першого вільного місця в бібліотеці макровизначень. Далі в бібліотеку записується весь рядок

директиви Macro. Після цього зчитуються наступні рядки макровизначення. Виконується лексичний аналіз та формується таблиця структури. Введений рядок записується в бібліотеку макровизначень, але таблиця лексем і таблиця структури не друкуються. Це відбувається, поки в бібліотеку не буде записано рядок з директивою Endm.

ЗАУВАЖЕННЯ. Що таке бібліотека макровизначень — вирішить програміст. Це може бути тимчасовий файл на диску (на час роботи компілятора). Або байтовий (char) масив, де використовуються деякі засоби для визначення кінця рядка (0 наприкінці, довжина на початку рядка тощо) і т.п.

На другому перегляді стандартно виконується лексичний аналіз та формується таблиця структури для визначення появи директиви Endm. Введені рядки розміщуються в лістингу. Більше ніяких дій не відбувається. Трансляція вхідних рядків у відповідний для другого перегляду спосіб розпочинається після появи директиви Endm.

Обробка макрокоманди

Макрокоманда обробляється стандартно, як і інші рядки, наприклад:

Таблиця Г.3. Таблиця лексем для TESTM2 ECX

№пп	Лексема	Довжина лексеми у символах	Тип лексеми
1	TESTM2	6	IDENTIFIER
2	ECX	3	REG32

Таблиця Г.4. Таблиця структури речення для TESTM2 ECX

Поле міток	Поле мнемокоду		Поле 1-го операнда	
№ лексеми поля	№ першої лексеми поля	Кількість лексем поля	№ першої лексеми операнда	Кількість лексем операнда
0	1	1	2	1

Формується тимчасова таблиця текстів фактичних параметрів з поля операнду та таблиці лексем. У даному прикладі таблиця текстів фактичних параметрів містить лише один фактичний параметр – текст <ECX>.

Формування макророзширення

Спочатку з макробібліотеки читається директива Macro. Формуються таблиця лексем та таблиця структури (друкувати їх не потрібно). Із цих таблиць формується таблиця формальних параметрів директиви Macro. У даному прикладі вона містить текст лише одного формального параметру – <T1>. Далі послідовно, до директиви Endm, виконуються наступні дії:

- зчитується черговий рядок із макробібліотеки

При лексичному аналізі, у разі появи ідентифікатора, він у першу чергу порівнюється з формальним параметром Macro. Якщо було співпадіння, то ідентифікатор формального параметра замінюється у введеному рядку на текст фактичного параметра із таблиці фактичних параметрів. Наприклад, до заміни – DEC T1. Після заміни – DEC ECX

- створюється (доповнюється) таблиця лексем
- створюється таблиця структури речення
- На другому етапі КР таблиця лексем та таблиця структури друкуються.

Наприклад: DEC ECX

Таблиця Г.5. Таблиця лексем для DEC ECX

№пп	Лексема	Довжина лексеми у символах	Тип лексеми
1	DEC	3	COMMAND
2	ECX	3	REG32

Таблиця Г.6. Таблиця структури речення для DEC ECX

Поле міток	Поле мнемокоду		Поле 1-го операнда	
№ лексеми поля	№ першої лексеми поля	Кількість лексем поля	№ першої лексеми операнда	Кількість лексем операнда
0	1	1	2	1

- На третьому та четвертому етапах все виконується аналогічно, окрім роздруківок таблиць.

ЗАУВАЖЕННЯ. Якщо формальні параметри відсутні, тоді ще простіше.

Додаток Д Опис найбільш уживаних команд реального режиму

Позначення, які використовуються при описі команд:

/r – вказує на те, що команда має байт mod r/m, в якому поле reg містить номер регістра даних;

/цифра (від 0 до 7) – вказує на те, що команда має байт mod r/m, в якому поле reg містить частину коду операції;

r/m8 r/m16 r/m32 – вказує на те, що команда має байт mod r/m, в якому поле r/m може містити або номер регістра даних відповідної розрядності, або команда формує ефективну адресу (зміщення в сегменті) пам'яті для даних відповідної розрядності;

ea16, ea32 – створювана процесором ефективна адреса (зміщення в сегменті) на основі байту\байтів режиму адресації (modR\m або modR\m і Sib) та зміщення в команді (displacement);

r8, r16, r32 – операнд в одному з регістрів розміром байт, слово чи подвійне слово;

ib, iw, id (або **imm8, imm16, imm32)** – безпосередній операнд розміром байт, слово чи подвійне слово;

cb, cw, cd – зміщення в команді;

+rb, +rw, +rd – байт коду операції команди в молодших трьох бітах містить номер регістра даних відповідної розрядності;

ptr16:16 – логічна адреса **– сегментна складова : зміщення всегменті**;

приймач – операнд пам'яті або регістр даних; в описі алгоритмів команд розуміємо як значення даних у пам'яті або в регістрі;

джерело - операнд пам'яті або регістр даних або константа. в описі алгоритмів команд розуміємо як значення даних у пам'яті або в регістрі, або значення константи.

Для опису стану ознак після виконання будь-якої команди будуть використовуватись наступні позначення:

1 – після виконання команди ознака встановлюється (дорівнює 1);

0 – після виконання команди ознака зкидається (дорівнює 0);

***** – значення ознаки залежить від результату роботи команди;

? – після виконання команди ознака не визначена;

позначення відсутнє – після виконання команди ознака не змінюється.

Опис найбільш уживаних команд представлено нижче.

Таблиця Д.1 – Команда ADC

ADC	**Додавання з перенесенням**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
		*				*	*	*	*	*
Код операції	***Структура команди***	***Приклад на асемблері***							***Такти***	
10 /r	**Adc r/m8, r8**	**Adc ah, al**							**1/3**	
11 /r	**Adc r/m16, r16**	**Adc word ptr sum, dx**							**1/3**	
11 /r	**Adc r/m32, r32**	**Adc ecx, ebx**							**1/3**	
12 /r	**Adc r8, r/m8**	**Adc ah, sum**							**1/2**	
13 /r	**Adc r16, r/m16**	**Adc di, ax**							**1/2**	
13 /r	**Adc r32, r/m32**	**Adc ecx,raznost**							**1/2**	
14 ib	**Adc al, imm8**	**Adc al, 4**							**1**	
15 iw	**Adc ax, imm16**	**Adc ax, 3fdh**							**1**	
15 id	**Adc eax, imm32**	**Adc eax, 37fd76fdh**							**1**	
80 /2 ib	**Adc r/m8, imm8**	**Adc byte ptr [si], 5**							**1/3**	
81 /2 iw	**Adc r/m16, imm16**	**Adc di,0abfh**							**1/3**	
81 /2 id	**Adc r/m32, imm32**	**Adc ecx, 0caaaah**							**1/3**	
83 /2 ib	**Adc r/m16, imm8**	**Adc bx, 0abh**							**1/3**	
83 /2 ib	**Adc r/m32, imm8**	**Adc dword ptr[edx], 0ah**							**1/3**	

Схема команди:	ADC *приймач, джерело*

Призначення: додавання двох цілих зі знаком або без знаку та значення cf.

Алгоритм роботи:

1) скласти два операнди та вміст cf;
2) помістити результат у перший операнд:
   приймач=приймач+джерело+cf;
3) в залежності від результату встановити ознаки.

Цілі операнди задаються у доповняльному коді. Результат додавання фактично не залежить від типу вхідних даних (беззнакові чи зі знаком у доповняльному коді). Інтерпретація результату здійснюється шляхом аналізу ознак.

ЗАУВАЖЕННЯ. Приймач та *джерело* команди ADC повинні мати однаковий розмір – байт, слово або подвійне слово. Якщо *джерело* являє собою безпосередні дані, то безпосередні дані у команді процесора можуть бути однобайтними незалежно від розміру *приймача* (див. код операції 83h). При цьому, однобайтні безпосередні дані при виконанні команди процесором знаково розширюються до необхідного розміру. TASM підтримує таку можливість оптимізації програм, яка передбачена в системі команд процесора, і генерують команди з однобайтними безпосередніми даними (з кодами операції 83h) за умови, шо безпосередні дані задані на Асемблері константами в діапазоні від – 128 до +127 (або від – 80h до +7fh).

Застосування: Команда ADC використовується при додаванні надвеликих двійкових беззнакових цілих чисел і надвеликих цілих чисел зі знаком, які подаються у доповняльному коді.

Таблиця Д.2 – Команда ADD

ADD	**Додавання**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
		*				*	*	*	*	*
Код операції	***Структура команди***	***Приклад на асемблері***							***Такти***	
04 ib	**ADD AL, imm8**	**Add al, 17**							**1**	
05 iw	**ADD AX, imm16**	**Add ax, 0AABh**							**1**	
05 id	**ADD EAX, imm32**	**Add eax, 0AAC23h**							**1**	
80 /0 ib	**ADD r/m8, imm8**	**Add bl, 5**							**1/3**	
81 /0 iw	**ADD r/m16, imm16**	**Add bx,0A4Fh**							**1/3**	
81 /0 id	**ADD r/m32, imm32**	**Add edx, 0CAAAAh**							**1/3**	
83 /0 ib	**ADD r/m16, imm8**	**Add word ptr [di], 7**							**1/3**	
83 /0 ib	**ADD r/m32, imm8**	**Add ecx, 0Fh**							**1/3**	
00 /r	**ADD r/m8, r8**	**Add ch, al**							**1/3**	
01 /r	**ADD r/m16, r16**	**Add word ptr m32, dx**							**1/3**	
01 /r	**ADD r/m32, r32**	**Add edx, ebx**							**1/3**	
02 /r	**ADD r8, r/m8**	**Add ch, sum**							**1/2**	
03 /r	**ADD r16, r/m16**	**Add si, ax**							**1/2**	
03 /r	**ADD r32, r/m32**	**Add edi,razn[ebx+esi*4]**							**1/2**	

Застосування: Команда ADC використовується при додаванні надвеликих двійкових беззнакових цілих чисел і надвеликих цілих чисел зі знаком, які подаються у доповняльному коді.

Схема команди:	ADD *приймач, джерело*

Призначення: додавання двох цілих зі знаком або без знаку.

Алгоритм роботи:

1) скласти вміст операндів *джерело* і *приймач*;
2) записати результат додавання в *приймач*;
3) в залежності від результату встановити ознаки.

Цілі зі знаком задаються у доповняльному коді. Результат додавання не залежить від типу вхідних даних (числа зі знаком чи без знаку). Інтерпретація результату при необхідності здійснюється шляхом аналізу ознак. Співвідношення розміру операндів – згідно зауважень до команди ADC.

Таблиця Д.3 – Команда AND

AND	**Логічне “І”**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
		*****				*****	*****	**?**	*****	**0**
Код операції	***Структура команди***	***Приклад на асемблері***							***Такти***	
1	***2***	***3***							***4***	
20 /r	**AND r/m8, r8**	**And [bx], ah**							**1/3**	
21 /r	**AND r/m16, r16**	**And cx, di**							**1/3**	
21 /r	**AND r/m32, r32**	**And memory, ecx**							**1/3**	
22 /r	**AND r8, r/m8**	**And ch, sum**							**1/2**	
23 /r	**AND r16, r/m16**	**And di, [si]**							**1/2**	
23 /r	**AND r32, r/m32**	**And ecx,raznost**							**1/2**	
24 ib	**AND AL, imm8**	**And al, 4**							**1**	
25 iw	**AND AX, imm16**	**And ax, 03FDh**							**1**	
25 id	**AND EAX, imm32**	**And eax, 23456789h**							**1**	
80 /4 ib	**AND r/m8, imm8**	**And byte ptr [bx], 5**							**1/3**	
81 /4 iw	**AND r/m16, imm16**	**And dx,0DBBh**							**1/3**	
81 /4 id	**AND r/m32, imm32**	**And ebx, 0CAAAAh**							**1/3**	
83 /4 ib	**AND r/m16, imm8**	**And cx, 0ABh**							**1/3**	
83 /4 ib	**AND r/m32, imm8**	**And edx, 0Ah**							**1/3**	

Схема команди:	AND *приймач, джерело*

Призначення: виконання порозрядного логічного множення.

Алгоритм роботи:

1) кожен біт результату дорівнює 1, якщо відповідні біти обох операндів дорівнюють 1, інакше – дорівнює 0;
2) записати результат операції в *приймач*;
3) установити ознаки.

Співвідношення розміру операндів – згідно зауважень до ADC.

Застосування: Команду зручно використовувати для примусового скидання в 0 довільної сукупності розрядів приймача без зміни вмісту решти розрядів, наприклад:

; нехай регістр dx містить значення 1110011001000111b=0e647h

And dx, 1111000000001111b

; після виконання команди dx=1110000000000111b=0e007h

Таблиця Д.4 – Команда BSF

BSF	**Сканування бітів уперед**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
							*			
Код операції	***Структура команди***	***Приклад на асемблері***						***Такти***		
0F BC	**BSF r16, r/m16**	**Bsf ax, word ptr [bx]**						**6-34/6-35**		
0F BC	**BSF r32, r/m32**	**Bsf eax, edx**						**6-42/6-43**		

Схема команди:	BSF *результат, джерело*

Призначення: пошук першого ненульового розряду (починаючи з молодших розрядів) в операнді *джерело*.

Алгоритм роботи:

1) перегляд бітів операнда *джерело*, починаючи з біта 0 і закінчуючи бітом 15/31, доти, поки не зустрінеться одиничний біт;

2) якщо зустрівся одиничний біт, то ознака zf встановлюється в 0, а в операнд *результат* записується номер позиції з цим бітом. Діапазон значень номера позиції залежить від розрядності операнда: для 16-розрядного операнда – 0...15; для 32-розрядного – 0...31;

3) якщо одиничних бітів немає, то ознака zf встановлюється в 1.

Застосування: Команду BSF використовують при роботі на бітовому рівні для визначення позиції в операнді *джерело* крайнього справа одиничного біта.

Наприклад, зсунемо вміст регістра bx вправо таким чином, щоб нульовий розряд містив 1:

```
.386
    Mov     bx,28h          ;bx=0100 1000b
    Bsf     cx,bh           ;cx=0003h
    Jz      null
    Shr     bx,cl           ;bx=0000 1001b
null:
```

Таблиця Д.5 – Команда BSR

BSR	**Сканування бітів назад**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
							*			
Код операції	***Структура команди***	***Приклад на асемблері***						***Такти***		
0F BD	**BSR r16, r/m16**	**Bsr ax, bitmask**						**7-39/7-40**		
0F BD	**BSR r32, r/m32**	**Bsr eax, edx**						**7-71/7-72**		

Схема команди:	BSR *результат, джерело*

Призначення: пошук першого (починаючи зі старших) ненульового розряду в операнді *джерело*.

Алгоритм роботи:

1) перегляд бітів операнда *джерело*, починаючи зі старшого біта за номером 15/31 і закінчуючи бітом за номером 0, доти, поки не зустрінеться одиничний біт;
2) якщо зустрівся одиничний біт, ознака zf встановлюється в 0, а в операнд *результат* записується номер позиції (*відлік номеру позиції здійснюється від молодших бітів*), де зустрівся найстарший одиничний біт. Діапазон значень номера позиції залежить від розрядності другого операнда: для 16-розрядного операнда – 0...15; для 32-розрядного – 0...31;
3) якщо одиничних бітів немає, ознака zf встановлюється в 1.

Застосування: Команду BSR використовують при роботі на бітовому рівні для визначення позиції крайніх зліва одиничних бітів.

Наприклад, зсунемо вміст регістра bx вліво таким чином, щоб старший одиничний біт значення в bx перемістився в 15-ту позицію:

```
.386
    Mov     bx,41h          ;bx=00000000_01000001b
    Bsr     cx,bh           ;cx=06h
    Jz      null
    Sub     cx,15           ;cx=-9
    Neg     cx              ;cx=9
    Shr     bx,cx           ;bx=10000010_00000000b
null:
```

Таблиця Д.6 – Команда BT

BT	**Перевірка біта**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
										*
Код операції	***Структура команди***	***Приклад на асемблері***							***Такти***	
0F A3	**BT r/m16, r16**	**Bt [di], ax**							**4/9**	
0F A3	**BT r/m32, r32**	**Bt edi, eax**							**4/9**	
0F BA /4 ib	**BT r/m16, imm8**	**Bt bx, 3**							**4**	
0F BA /4 ib	**BT r/m32, imm8**	**Bt edx, 17**							**4**	

Схема команди:	BT *джерело, позиція*

Призначення: копіювання значення заданого біта в ознаку cf.

Алгоритм роботи:

1) Якщо операнд *джерело* є регістром, то визначити *номер_розряду* = *позиція* ***mod*** 16 для 16-розрядних регістрів або *номер_розряду* = *позиція* ***mod*** 32 для 32 регістрів. Значення розряду операнда *джерело* за отриманим номером розряду скопіювати в ознаку cf.
2) Якщо операнд *джерело* є адресою пам'яті, а операнд *позиція* – безпосередні дані, то визначити *номер_розряду* = *позиція* ***mod*** 16 для 16 -розрядних даних або *номер_розряду* = *позиція* ***mod*** 32 для 32-розрядних даних (нові версії ассемблера добавляють при трансляції до зміщення в команді значення 2*(*позиція* ***div*** 16) або 4*(*позиція* ***div*** 32). Значення розряду операнда *джерело* за отриманим номером розряду скопіювати в ознаку cf.
3) Якщо операнд *джерело* є адресою пам'яті, а операнд *позиція* – регістр, в якому міститься додатнє число у доповняльному коді, то сформувати *нова_адреса* = *джерело* + *(вміст_регістру* ***div*** 8) та *номер_розряду* = *(вміст_регістру* ***mod*** 8). Вміст розряду за отриманим номером *номер_розряду* байта пам'яті за отриманою адресою *нова_адреса* скопіювати в ознаку cf.
4) Якщо операнд *джерело* є адресою пам'яті, а операнд *позиція* – регістр, в якому знаходиться від'ємне число у доповняльному коді, то сформувати *нова_адреса* = *джерело* + *(вміст_регістру* ***div*** 8) з округленням до цілого у напрямку - ∞ та *номер_розряду* = *7* + *(вміст_регістру* ***mod*** 8). Вміст розряду за отриманим номером *номер_розряду* байта пам'яті за отриманою адресою *нова_адреса* скопіювати в ознаку cf.

Застосування: Команду BT використовують для визначення конкретного розряду даних, заданих операндом *джерело*. Наприклад, за допомогою цієї команди просто реалізувати умовну передачу управління за значенням довільного розряду регістра загального призначення.

```
.386
    Bt      edx,18   ;копіювання вмісту 18-го розряду edx в ознаку cf
    Jc      m1       ;перейти на m1, якщо біт, що перевіряється, дорівнює 1
```

Використання команди BT m32, r32 дозволяє аналізувати бітові поля в пам'яті розміром до 4Гбіт.

Таблиця Д.7 – Команда BTC

BTC	**Перевірка та інверсія біта**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
										*
Код операції	***Структура команди***	***Приклад на асемблері***						***Такти***		
0F BB	**BTC r/m16, r16**	**Btc [di], ax**						**7/13**		
0F BB	**BTC r/m32, r32**	**Btc [edi], eax**						**7/13**		
0F BA /7 ib	**BTC r/m16, imm8**	**Btc [bx], 05**						**7/8**		
0F BA /7 ib	**BTC r/m32, imm8**	**Btc ebx, 02**						**7/8**		

Схема команди:	BTC *джерело, позиція*

Призначення: копіювання розряду операнда *джерело* в ознаку cf з наступною інверсією цього розряду.

Алгоритм роботи:

1) виконати команду BT *джерело, позиція;*
2) інвертувати розряд даних, визначений за операндами *джерело* та *позиція* (див. алгоритм команди BT).

Застосування: Команду BTC доцільно використовувати для інверсії заданого розряду даних. При цьому попереднє значення розряду може бути використане для розгалуження в програмі за допомогою команди jc або jnc.

```
.386
    Mov     bx,4abh         ;інверсія біту 8 і перевірка його
                            ;попереднього стану:
    Btc     bx,8            ;bx=5abh, cf=0
    Jnc     @1
```

Таблиця Д.8 – Команда BTR

BTR	**Перевірка та скидання біта**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
										*
Код операції	***Структура команди***	***Приклад на асемблері***						***Такти***		
0F B3	**BTR r/m16, r16**	**Btr [di], ax**						**7/13**		
0F B3	**BTR r/m32, r32**	**Btr [edi], eax**						**7/13**		
0F BA /6 ib	**BTR r/m16, imm8**	**Btr [bx+8], 05**						**7/8**		
0F BA /6 ib	**BTR r/m32, imm8**	**Btr [ebx], 02**						**7/8**		

Схема команди:	BTR *джерело, позиція*

Призначення: копіювання вмісту заданого розряду в ознаку cf з наступним записом у цей розряд 0.

Алгоритм роботи:

1) виконати команду BT *джерело, позиція;*
2) записати 0 в розряд даних, визначений за операндами *джерело* та *позиція* (див. алгоритм команди BT).

Застосування: Команде BTR доцільно використовувати для запису 0 в окремий розряд даних (скидання розряду). При цьому попереднє значення розряду може бути використане для розгалуження в програмі за допомогою команд JC або JNC

```
.386
    Mov     bx,5abh         ;скидання біту 8 і перевірка його
                            ;попереднього стану:
    Btc     bx,8            ;bx=4abh, cf=1
    Jnc     @1
```

Таблиця Д.9 – Команда BTS

BTS	**Перевірка і встановлення біта**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
										*
Код операції	***Структура команди***	***Приклад на асемблері***						***Такти***		
0F AB	**BTS r/m16, r16**	**Bts [di], bp**						**7/13**		
0F AB	**BTS r/m32, r32**	**Bts [edi], ebx**						**7/13**		
0F BA /5 ib	**BTS r/m16, imm8**	**Bts word ptr [bx+si], 05**						**7/8**		
0F BA /5 ib	**BTS r/m32, imm8**	**Bts dword hnr [edi+ebx], 02**						**7/8**		

Схема команди:	BTS *джерело, позиція*

Призначення: копіювання вмісту заданого розряду в ознаку cf з наступним записом в цей розряд 1.

Алгоритм роботи:

1. виконати команду BT *джерело, позиція*
2. записати 1 в розряд даних, визначений операндами *джерело* та *позиція* (див. алгоритм команди BT).

Застосування: Команду BTS доцільно використовувати для запису 1 в окремий розряд даних (встановлення розряду). При цьому попереднє значення розряду може бути використане для розгалуження в програмі за допомогою команд JC або JNC.

```
.386
    Mov     bx,4abh         ;встанолення біту 8 і перевірка його
                            ;попереднього стану :
    Btc     bx,8            ;bx=4abh, cf=1
    Jnc     @1
```

Таблиця Д.10 – Команда CALL

CALL	**Виклик процедури**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
Код операції	***Структура команди***	***Опис типу виклику. Приклад на асемблері***						***Такти***		
E8 cw	**CALL rel16**	**Внутрішньосегментний відносний, зміщення вказується відносно адреси наступної команди Call near ptr p1**						**1**		
FF /2	**CALL r/m16**	**Внутрішньосегментний посередній або внутрішньосегментний посередній регістровий Call word ptr [si] або Call ax**						**2**		
9A cd	**CALL ptr16:16**	**Міжсегментний прямий, вказується 4 байтна логічна адреса Call far ptr p1**						**4**		
FF /3	**CALL m16:16**	**Міжсегментний посередній Call dword ptr [si]**						**5**		

Призначення: виклик процедури.

Алгоритм роботи в реальному режимі:

1) При внутрішньосегментному виклику

a. записати в стек вміст регістра IP;

b. якщо відносна адресація (код операції 0E8h), то IP:=IP+rell16;

c. якщо посередня регістрова адресація (код операції FF /2, mod=11), то IP:=r16;

d. якщо посередня адресація (код операції FF /2, mod ≠ 11), то IP:=m16;

2) При міжсегментному виклику

a. записати в стек вміст регістра CS;

b. записати в стек вміст регістра IP;

c. якщо пряма адресація (код операції 09ah), то CS:IP = ptr16:16;

d. якщо посередня адресація (код операції FF /3, mod ≠ 11), то CS:IP:=m32.

ЗАУВАЖЕННЯ. Теоретично команда CALL в реальному режимі може виконуватись з використанням 32-розрядного зміщення, але фактичне значення 32-розрядного зміщення не повинно перевищувати 0FFFFh.

Застосування: Команда CALL застосовується для виклику процедур. Адреса повернення (адреса наступної після CALL команди в ОЗП) записується в стек, а управління безумовно передається на початок процедури. Повернення з процедури за адресою, яка записана в стек, відбувається шляхом виконання команди RET. На мові Асемблера процедура розпочинається директивою PROC і закінчується директивою ENDP.

Команди CALL в процесорах 80x86 діляться на внутрішньосегментні та міжсегментні. Серед внутрішньосегментних команд CALL найчастіше застосовується команда CALL з відносною адресацією, наприклад:

```
P1      proc
        ...
        ret
P1      endp
...
Call    P1
...
```

Асемблер генерує для такої команди CALL адресну частину у доповняльному коді, виходячи із формули $A_к=З_{пр} - З_{нк}$, де $A_к$ – адресна частина команди CALL, $З_{пр}$ – зміщення процедури, $З_{нк}$ – зміщення наступної команди.

Команда CALL з непрямою (посередньою) адресацією застосовується у випадках, коли для обробки інформації існують декілька процедур, а

виклик конкретної процедури залежить від даних, які визначаються тільки при виконанні програми, наприклад, вводяться із зовнішнього пристрою:

```
data     segment
adr_proc dw   P0
         dw   P1
  ...
         dw   p9
data     ends
code     segment
P0       proc
   ...
         ret
P0       endp
P1       proc
   ...
         ret
P1       endp
....
P9       proc
   ...
         ret
P9       endp
...
         Mov   ah,0          ;ввести з клавіатури одну із десяткових цифр
         Int   16            ;ah – скан-код клавіші, al – ASCII  код символу
         Cmp   al,30h ;
         Jc    error         ;
         Cmp   al,3ah        ;передача управління на мітку error
         Jnc   error         ;якщо введена не десяткова цифра
         Sub   al,30h        ;al – число від 0 до 9
         Xor   bx,bx
         Mov   bl,al
         Shl   bx,1          ;bx – зміщення адреси процедури в масиві adr_proc
         Call  adr_proc[bx]
....
  code   ends
```

ЗАУВАЖЕННЯ. Виклик процедури за непрямою адресою є викликом за вказівником на процедуру, що широко застосовується в C++ і в системному програмуванні взагалі.

Таблиця Д.11 – Команда CBW

CBW	**Перетворити байт в слово**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
CWDE	**Перетворити слово в подвійне слово**									
Код операції	***Структура команди***	***Приклад на асемблері***						***Такти***		
98	**CBW**	**Cbw**						**3**		
98	**CWDE**	**Cwde**						**3**		

Схема команди:	CBW
	CWDE

Призначення: розширення операнда зі знаком.

Алгоритм роботи:

CBW – при роботі команда використовує регістри al і ah:

1) аналіз знакового біта регістра al:
   a. якщо знаковий біт al=0, то ah=00h;
   b. якщо знаковий біт al=1, то ah=0ffh.

CWDE – при роботі команда використовує регістри ax і eax:

2) аналіз знакового біта регістра ax:
   a. якщо знаковий біт ax=0, то скинути всі розряди старшого слова eax в 0;
   b. якщо знаковий біт ax=1, то установити всі розряди старшого слова eax в 1.

Застосування: команди використовуються для приведення операндів до потрібної розмірності з урахуванням знаку. Така необхідність може, зокрема, виникнути при програмуванні арифметичних операцій.

```
      .386                          ;тільки для cwde, cwd була для i8086
       Mov   ebx,10fecd23h
; нехай ax=1111 1111 1111 1101 (-3)
       Cwde                          ;eax=1111 1111 1111 1111 1111 1111 1111 1101
```

Add ebx, eax ; команда add ebx,ax відсутня

Таблиця Д.12 – Команда CLC

CLC	**Очистити ознаку перенесення**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
										0
Код операції	***Структура команди***	***Приклад на асемблері***						***Такти***		
F8	**CLC**	**Clc**						**2**		

Схема команди:	CLC

Призначення: скидання ознаки перенесення cf.

Алгоритм роботи: cf = 0.

Застосування: Біт cf використовується як джерело даних в ряді команд, наприклад в командах циклічного зсуву, в командах ADC та SBB. У зв’язку з цим, команда CLC може використовуватись для попереднього визначення цього біту.

Таблиця Д.13 – Команда CLD

CLD	**Очистити ознаку напрямку**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
			0							
Код операції	***Структура команди***	***Приклад на асемблері***						***Такти***		
FC	**CLD**	**Cld**						**2**		

Схема команди:	CLD

Призначення: скидання в нуль ознаки напрямку df.

Алгоритм роботи: df = 0

Застосування: Необхідність використання команди виникає при роботі з ланцюговими командами. Нульове значення ознаки df змушує мікропроцесор при виконанні ланцюгових операцій інкрементувати вміст регістрів si, di (рядки обробляються у прямому, а не зворотному напрямку).

Таблиця Д.14 – Команда CLI

CLI	**Заборонити зовнішні переривання**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
				0						
Код операції	***Структура команди***	***Приклад на асемблері***							***Такти***	
FA	**CLI**	**Cli**							**7**	

Схема команди:	CLI

Призначення: заборонити переривання від зовнішніх пристроїв.

Алгоритм роботи: df = 0

Застосування: Команда CLI задається перед тими фрагментами програми, для виконання яких існують жорсткі обмеження у часі, або при виконанні яких використовуються дані, що змінюються в результаті виконання процедури обробки переривання. Запити на переривання від зовнішніх пристроїв запам’ятовуються в контролері переривань, тому в кінці критичного фрагменту програми ставлять команду STI, яка відновлює дозвіл на переривання.

Таблиця Д.15 – Команда CMC

CMC	**Очистити ознаку перенесення**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
										*
Код операції	***Структура команди***	***Приклад на асемблері***							***Такти***	
F5	**CMC**	**Cmc**							**2**	

Схема команди:	CMC

Призначення: зміна значення ознаки переносу cf на протилежне.

Алгоритм роботи: cf = not cf

Застосування: Біт cf використовується як джерело даних в ряді команд (наприклад, в командах циклічного зсуву, в командах ADC та SBB).

```
proc1   proc
…
        Cmc
…
proc1   endp
…
call    proc1
        Jc      m1          ;якщо cf=1, то перехід на m1
…
m1:
```

Таблиця Д.16 – Команда CMP

CMP	**Порівняти два операнди**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
		*				*	*	*	*	*
Код операції	***Структура команди***	***Приклад на асемблері***					***Такти***			
3C ib	**CMP AL, imm8**	**Cmp al, 17**					**1**			
3D iw	**CMP AX, imm16**	**Cmp ax,0AABh**					**1**			
3D id	**CMP EAX, imm32**	**Cmp eax,0AAC23h**					**1**			
80 /7 ib	**CMP r/m8, imm8**	**Cmp bl, 5**					**1/2**			
81 /7 iw	**CMP r/m16, imm16**	**Cmp bx, 0A4Fh**					**1/2**			
81 /7 id	**CMP r/m32, imm32**	**Cmp edx, 0CAAAAh**					**1/2**			
83 /7 ib	**CMP r/m16, imm8**	**Cmp word ptr [di], 7**					**1/2**			
83 /7 ib	**CMP r/m32, imm8**	**Cmp ecx, 0Fh**					**1/2**			
38 /r	**CMP r/m8, r8**	**Cmp ch, al**					**1/2**			
39 /r	**CMP r/m16, r16**	**Cmp word ptr m32, dx**					**1/2**			
39 /r	**CMP r/m32, r32**	**Cmp edx, ebx**					**1/2**			
3A /r	**CMP r8, r/m8**	**Cmp ch, sum**					**1/2**			
3B /r	**CMP r16, r/m16**	**Cmp si, ax**					**1/2**			
3B /r	**CMP r32, r/m32**	**Cmp edi, raznost**					**1/2**			

Схема команди:	CMP *операнд1, операнд2*

Призначення: порівняння двох операндів.

Алгоритм роботи:

1) виконати віднімання (*операнд1-операнд2*);

2) залежно від результату встановити ознаки, *операнд1* і *операнд2* не змінювати (тобто результат не запам'ятовується).

Застосування: команда використовується для порівняння двох операндів шляхом віднімання, при цьому операнди не змінюються. За результатами виконання команди встановлюються ознаки. Команда CMP застосовується з командами умовного переходу jcc.

```
len     equ   10
…
Cmp     ax,len
Jnz     m1          ;перехід якщо (ax)<>len
Jmp     m2          ;перехід якщо (ax)=len
```

Таблиця Д.17 – Команди CMPS

CMPS **CMPSB** **CMPSW** **CMPSD**	**Порівняти рядкові операнди**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
		*				*	*	*	*	*
Код операції	***Структура команди***	***Приклад на асемблері***							***Такти***	
A6	**CMPS m8, m8**	**Cmps byte ptr dst, gs:byte ptr src**							**5**	
A7	**CMPS m16, m16**	**Cmps word ptr dst, word ptr src**							**5**	
A7	**CMPS m32, m32**	**Cmps dword ptr dst, dword ptr src**							**5**	
A6	**CMPSB**	**Cmpsb**							**5**	
A7	**CMPSW**	**Cmpsw**							**5**	
A7	**CMPSD**	**Cmpsd**							**5**	

Схема команди:	CMPS *приймач,джерело*
	CMPSB
	CMPSW
	CMPSD

Призначення: порівняння елементів двох рядків (послідовностей, ланцюжків, масивів) у пам'яті.

Алгоритм роботи:

1) виконати віднімання операндів (*джерело* - *приймач*), логічні адреси елементів попередньо повинні бути завантажені:

a. адреса *джерела* - у пари регістрів ds:esi/si;

b. адреса *приймача* - у пари регістрів es:edi/di;

2) залежно від результату віднімання встановити ознаки:

a. якщо чергові елементи ланцюжків не однакові, то zf=0;

b. якщо чергові елементи ланцюжків однакові, то zf=1;

3) залежно від стану ознаки df змінити значення регістрів esi/si і edi/di:

a. якщо df=0, то збільшити вміст цих регістрів на довжину елемента послідовності (тобто ланцюжки порівнюються з початку);

b. якщо df=1, то зменшити вміст цих регістрів на довжину елемента послідовності (тобто ланцюжки порівнюються з кінця);

4) при наявності префікса повторення виконати обумовлені ним дії (див. префікси repe/repne).

При трансляції команд CMPSW та CMPSD Асемблер генерує префікс зміни розрядності даних (код 66h), якщо розрядність даних за замовчуванням не відповідає мнемокоду команди. Розрядність адрес для команд CMPSB, CMPSW та CMPSD (відповідно до використання регістра SI чи ESI) встановлюється тільки за замовчуванням. При трансляції команди CMPS Асемблер може згенерувати префікс зміни розрядності даних, префікс зміни розрядності адрес і один із префіксів заміни сегменту. Нехай за замовчуванням встановлені 16-розрядні дані та адреси (атрибут USE16 в директиві SEGMENT), тоді в результаті трансляції машинної інструкції

Cmps dword ptr [edi], gs:dword ptr [edi]

Асемблер згенерує 4 байти:

67h| 66h| 65h: 0A7h, де 67h – префікс зміни розрядності адрес, 66h – префікс зміни розрядності даних, 65h – префікс заміни сегменту, 0A7h – код команди CMPSW. Таким чином, поле операндів в машинній інструкції CMPS фактично використовується для генерування префіксів.

Застосування: Команди без префіксів повторення здійснюють просте порівняння двох елементів у пам'яті. Розміри порівнюваних елементів залежать від команди, яка застосовується. Для адресації *приймача* обов'язково повинен використовуватися регістр es, а для адресації *джерела* можна робити заміну сегмента з використанням відповідного префікса заміни сегменту.

Для використання цих команд при порівнянні послідовностей елементів, що мають розмірність байт, слово, подвійне слово, необхідно використовувати один із префіксів: repe чи repne. Префікс repe змушує циклічно виконуватися команди порівняння доти, поки вміст регістра ecx/cx не дорівнюватиме нулю або поки zf=1 (тобто поки рядки співпадають). Префікс repne змушує циклічно робити порівняння доти, поки не буде досягнутий кінець ланцюжка (ecx/cx=0) або поки zf=0 (тобто поки рядки відрізняються).

```
.data
obl1    db    'Рядок для порівняння'
obl2    db    'Рядок для порівняння'
a_obl1  dd    obl1
a_obl2  dd    obl2
.code
...
Cld                 ;перегляд ланцюжка у напрямку зростання адрес
Mov     cx,20       ;довжина ланцюжка
Lds     si,a_obl1   ;адреса джерела в парі ds:si
Les     di,a_obl2   ;адреса призначення в парі es:di
Repe    cmpsb       ;порівнювати, поки рівні
Jnz     m1          ;якщо не кінець ланцюжка, то
                    ;зустрілися різні  елементи
;дії, якщо ланцюжки співпадають
...
m1:                 ;дії, якщо ланцюжки не співпадають
Lodsb               ;al – елемент, який не співпав
```

ЗАУВАЖЕННЯ. У реальному режимі використовуються лише молодші 16 розрядів регістрів ESI та EDI.

Таблиця Д.18 – Команди CWD і CDQ

CWD	**Перетворити слово у подвійне слово**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
CDQ	**Перетворити подвійне слово у зчетверене слово**									
Код операції	***Структура команди***	***Приклад на асемблері***						***Такти***		
99	**CWD**	**Cwd**						**2**		
99	**CDQ**	**Cdq**						**2**		

Призначення: знакове розширення вмісту регістра AX/EAX з використанням регістра DX/EDX.

Алгоритм роботи:

1) якщо старший (знаковий) розряд регістра AX/EAX дорівнює 0, то DX/EDX = 0.

2) якщо старший (знаковий) розряд регістра AX/EAX дорівнює 1, то DX/EDX = 0ffffh/0ffffffffh.

Застосування: Команда CWD/CDQ використовується для розширення значення знакового біта в регістрі ax/eax в біти регістра dx/edx. Операцію, зокрема, можна використовувати для підготовки до операції ділення, для якої розмір діленого повинен бути в два рази більше розміру дільника.

```
Mov     ax,25
Mov     bx,4
Cwd
Div     bx
.386
delimoe dd      ...
delitel  dd     ...
...
Mov     eax,delimoe
Cdq
Idiv    delitel         ;частка в eax, залишок у edx
```

Таблиця Д.19 – Команда DAA

DAA	**Десяткова корекція AL після додавання**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
		?				*****	*****	*****	*****	*****
Код операції	***Структура команди***	***Приклад на асемблері***						***Такти***		
27	**DAA**	**Daa**						**3**		

Призначення: корекція результату додавання двох упакованих BCD-чисел.

В результаті двійкового додавання двох однорозрядних десяткових чисел, які подаються як чотирьохрозрядне (тетрада) двійкове число, можливі наступні випадки:

1) Сума менша ніж 10, молодша тетрада після двійкового додавання матиме значення менше ніж 10, корекція не потрібна.

2) Сума більша ніж 9, але менша ніж 16, молодша тетрада після двійкового додавання матиме значення більше 9. У цьому випадку, перенесення в наступну тетраду при двійковому додаванні не відбулося, тому необхідно від суми відняти 10 і забезпечити перенесення в наступну тетраду. Це автоматично відбувається шляхом додавання 6 (6 – це доповняльний (до 16) код від'ємного числа 10).

3) Сума цифр більша ніж 15, молодша тетрада після двійкового додавання матиме значення менше 3. У цьому випадку, при двійковому додаванні було перенесення 16 в старшу тетраду, а потрібно лише 10. Тому необхідно до молодшої тетради додати 6.

Алгоритм роботи:

1) Якщо молодша тетрада регістра al має значення більше 9 або ознака af=1, то додати до al 6.

   a. якщо перед додаванням 6 ознака cf=1, то вона не змінюється;

   b. якщо перед додаванням 6 ознака cf=0, то вона встановлюється за результатом додавання 6.

2) Якщо після виконання п.1) старша тетрада регістра al має значення більше 9 або ознака cf=1, то додати до al 60h.

a. якщо перед додаванням 60h ознака cf=1, то вона не змінюється;

b. якщо перед додаванням 60h ознака cf=0, то вона встановлюється за результатом додавання 60h.

Застосування: Команду застосовують після двійкового додавання двох упакованих BCD-чисел, щоб сформувати правильне двозначне десяткове число. Після команди daa стан ознаки cf визначає перенесення в старший десятковий розряд.

```
Mov   al,69h      ;69h - упаковане BCD-число
Mov   bl,74h      ;74h - упаковане BCD-число
Add   al,bl       ;al=0dbh, cf=0, af=0
Daa               ;cf=1, al=43h

Mov   al,19h
Mov   bl,89h
Add   al,bl       ;al=0a2h, cf=0,af=1
Daa               ;al=08h, cf=1
```

Таблиця Д.20 – Команда DAS

DAS	**Десяткове перетворення AL після віднімання**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
		?				*****	*****	*****	*****	*****
Код операції	***Структура команди***	***Приклад на асемблері***						***Такти***		
2F	**DAS**	**Das**						**3**		

Призначення: корекція віднімання двох BCD-чисел в упакованому форматі.

Алгоритм роботи:

1) Якщо (al and 0fh) > 9 або af=1 то

a. al=al-6;

b. Якщо ознака cf=1, то залишити її без змін;

c. Якщо ознака cf=0, то встановити її значення за результатом п.1a);

d. af=1;

2) Якщо (al and 0fh) < 10 та af=1 то af=0;

3) Якщо після виконання п.1 al>9fh або cf=1 то

a. al=al-60h

b. Якщо ознака cf=1, то залишити її без змін;

c. Якщо ознака cf=0, то встановити її значення за результатом п.3a);

d. cf=1;

4) Якщо після виконання п.1 al<0a0h та cf=1, то cf=0.

Застосування: Команду DAS варто застосовувати після двійкового віднімання двох упакованих BCD-чисел з метою корегування результату та отримання правильного BCD число. Після команди DAS варто аналізувати стан ознаки cf. Якщо cf дорівнює 1, то це говорить про те, що була позика одиниці із наступного старшого розряду.

```
Mov     al,95h          ;al=95h
Mov     ah,86h          ;ah=86h
Sub     al,ah           ;al=al-ah=95h-86h=0fh, - не BCD-число, af=1,cf=0
Das                     ;al=09h - результат скорегований, af=1,cf=0
```

Таблиця Д.21 – Команда DEC

DEC	**Декремент**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
		*				*	*	*	*	
Код операції	***Структура команди***	***Приклад на асемблері***						***Такти***		
FE /1	**DEC r/m8**	**Dec al**						**1/3**		
FF /1	**DEC r/m16**	**Dec word ptr [di]**						**1/3**		
FF /1	**DEC r/m32**	**dec dword ptr [ebx]**						**1/3**		
48+rw	**DEC r16**	**dec dx**						**1**		
48+rd	**DEC r32**	**dec ecx**						**1**		

Схема команди:	DEC *операнд*

Призначення: зменшення на 1 значення операнда в пам'яті або регістрі.

Застосування: Команда DEC використовується для зменшення значення байта, слова, подвійного слова в пам'яті або регістрі на 1. Відзначимо, що ознака cf не змінюється. Це дозволяє використовувати команду DEC для зменшення індексу циклу, а ознаку cf застосовувати як міжітераційну ознаку. Наприклад:

```
Mov     bx,1000         ;індекс циклу
Clc
fordown:
  Mov   ax,[bx]
  Jc    label

...
label:
  Cmp   ax,dx           ;формування значення cf
  Dec   bx              ;cf не змінюється, zf-встановлюється по результату
  Jnz   fordown
```

Таблиця Д.22 – Команда DIV

DIV	**Ділення цілих беззнакових чисел**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
		?				**?**	**?**	**?**	**?**	**?**

Код операції	***Структура команди***	***Приклад на асемблері***	***Такти***
F6 /6	**DIV AL, r/m8**	**Div cl**	**17**
F7 /6	**DIV AX, r/m16**	**Div word ptr [bx]**	**25**
F7 /6	**DIV EAX, r/m32**	**Div edi**	**41**

Схема команди:	DIV *дільник*

Призначення: виконання операції ділення двох беззнакових чисел.

Алгоритм роботи: Для команди необхідно задати два операнди: ділене та дільник. Ділене задається неявно, і його розмір залежить від розміру дільника, що вказується в команді:

1) якщо дільник має розмір байт, то ділене повинно знаходитись в регістрі ax. Після операції частка міститься в al, а залишок – в ah;

2) якщо дільник має розмір слово, то ділене повинно знаходитись в парі регістрів dx:ax, причому молодша частина діленого – в ax. Після операції частка міститься в ax, а залишок – у dx;

3) якщо дільник має розмір подвійне слово, то ділене повинно знаходитись в парі регістрів edx:eax, причому молодша частина діленого – в eax. Після операції частка міститься в eax, а залишок – у edx.

4) якщо значення частки перевищує розмір регістра призначення або дільник = 0, тоді виникає внутрішнє переривання за 0-м вектором.

Застосування: Особливістю команди div є формування не тільки частки, а й залишку від ділення цілих чисел, що дозволяє, наприклад, організувати ділення надвеликих (багатобайтних) цілих чисел на одно-, дво- або чотирьохбайтні числа за алгоритмом ділення десяткових чисел у “стовпчик”.

Наприклад, нехай регістр SI містить зміщення старшого байта діленого, регістр DI – старшого байта частки, регістр BL містить дільник, регістр CX – кількість байт діленого. Наступна послідовність команд виконує ділення багатобайтного числа на однобайтне:

```
  Mov   ah,0          ;гарантія відсутності “ділення на 0” на першому кроці
m1:
  Mov   al,[si]       ;al - черговий байт діленого

        ;оскільки ah < bl і al < 256, то ((ah*256+al) div bl )<256
        ;тобто “ділення на 0” неможливе
; ділення ax на bl
  Div   bl
                      ;al – черговий байт частки, ah – залишок
;записати черговий байт частки
  Mov   [di],al
  Dec   di
  Dec   si
  Loop  m1
```

Таблиця Д.23 – Команда IDIV

IDIV	**Ділення цілих чисел зі знаком**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
		?				**?**	**?**	**?**	**?**	**?**

Код операції	***Структура команди***	***Приклад на асемблері***	***Такти***
F6 /7	**IDIV AL, r/m8**	**Idiv dl**	**22**
F7 /7	**IDIV AX, r/m16**	**Idiv cx**	**30**
F7 /7	**IDIV EAX, r/m32**	**Idiv esi**	**46**

Схема команди:	IDIV *дільник*

Призначення: Ділення двох цілих чисел зі знаком.

Алгоритм роботи: Для команди необхідно задати два операнди: ділене та дільник. Ділене задається неявно і його розмір залежить від розміру дільника, що вказується в команді:

1) якщо дільник має розмір байт, то ділене повинно знаходитись в регістрі ax. Після операції частка міститься в al, а залишок – в ah;

2) якщо дільник має розмір слово, то ділене повинно знаходитись в парі регістрів dx:ax, причому молодша частина діленого – в ax. Після операції частка міститься в ax, а залишок – у dx;

3) якщо дільник має розмір подвійне слово, то ділене повинно знаходитись в парі регістрів edx:eax, причому молодша частина діленого – в eax. Після операції частка міститься в eax, а залишок – у edx.

4) якщо значення частки перевищує розмір регістра призначення або дільник = 0, тоді виникає внутрішнє переривання за 0-м вектором.

5) знаковий розряд частки формується в результаті операції XOR знакових розрядів діленого і дільника. Залишок завжди має знак діленого.

Застосування: Команда застосовується для ділення цілих чисел, які подані у доповняльному коді, старший розряд чисел – знаковий.

```
Mov     ax,-1045        ;ділене ax=0fbebh
Mov     bx,520          ;дільник bx=208h
Cwd                     ;розширення діленого в dx:ax, dx=0ffffh
Idiv    bx      ;частка в ax, (ax = -2 = 0fffdh) залишок у dx (dx = -5=0fffbh)
```

Таблиця Д.24 – Команда IMUL

IMUL	**Множення цілих чисел зі знаком**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
		*****				**?**	**?**	**?**	**?**	**?**

Код операції	***Структура команди***	***Приклад на асемблері***	***Такти***
F6 /5	**Imul r/m8**	**Imul byte ptr[si]**	**11**
F7 /5	**Imul r/m16**	**Imul dx**	**11**
F7 /5	**Imul r/m32**	**Imul dword ptr[si]**	**11**
0F AF /r	**Imul r16,r/m16**	**Imul word ptr[si]**	**10**
0F AF /r	**Imul r32,r/m32**	**Imul eax,dword ptr[edx*8]**	**10**
6B /r ib	**Imul r16,r/m16, imm8**	**Imul ax,[bx],6**	**10**
6B /r ib	**Imul r32,r/m32, imm8**	**Imul ecx,esi,11**	**10**
6B /r ib	**Imul r16,imm8**	**Imul si,6**	**10**
6B /r ib	**Imul r32,imm8**	**Imul edi,23**	**10**
69 /r iw	**Imul r16,r/m16, imm16**	**Imul ax,dx,166h**	**10**
69 /r id	**Imul r32,r/m32, imm32**	**Imul ecx,esi,112233h**	**10**
69 /r iw	**Imul r16,imm16**	**Imul si,166h**	**10**
69 /r id	**Imul r32,imm32**	**Imul edx,666777h**	**10**

Схема команди:	IMUL *множ_1* IMUL l *множ_1,множ_2* IMUL *рез-т,множ_1,множ_2*

Призначення: операція множення двох цілих чисел зі знаком.

Алгоритм роботи: Алгоритм роботи команди залежить від кількості операндів. Команди з одним операндом вимагають явної вказівки місця розташування тільки одного з множників, що може бути розташований у

пам'яті чи регістрі. Місце розташування другого множника фіксоване і залежить від розміру першого множника:

1) якщо операнд, зазначений у команді, має розмір байт, то другий множник розташовується в al;
2) якщо операнд, зазначений у команді, має розмір слово, то другий множник розташовується в ax;
3) якщо операнд, зазначений у команді, має розмір подвійне слово, то другий множник розташовується в eax.

Результат множення для команди з одним операндом міститься у чітко визначеному регістрі (регістрах):

1) при множенні байтів результат міститься в ax;
2) при множенні слів результат міститься в парі dx:ax;
3) при множенні подвійних слів результат міститься в парі edx:eax.

Команди з двома і трьома операндами однозначно визначають розташування результату і множників у такий спосіб:

1) у команді з двома операндами перший операнд визначає місце розташування першого співмножника. На його місце згодом буде записаний результат. Другий операнд визначає місце розташування другого множника: *множ_1 = множ_1 * множ_2* ;

2) у команді з трьома операндами перший операнд визначає місце розташування результату, другий операнд – місце розташування першого множника, третій операнд може бути лише безпосередньо заданим значенням розміром байт, слово чи подвійне слово:

*множ_1 = множ_2 * множ_3* .

Команда IMUL встановлює в нуль ознаки of і cf, якщо розмір результату відповідає регістру призначення. Якщо ці ознаки відмінні від нуля, то це означає, що результат занадто великий для відведеного йому регістром призначення розміру, і необхідно вказати більший за розміром

регістр для успішного завершення операції множення. Конкретними умовами скидання ознакаів of і cf в нуль є наступні умови:

1) для одноопераднних команд IMUL регістри ax/dx/edx є знаковими розширеннями регістрів al/ax/eax;

2) для двох- та трьохоперандних команд IMUL для розміщення результату множення достатньо розмірності регістрів, зазначених як множ_1;

ЗАУВАЖЕННЯ. Для мікропроцесора i8086 можлива тільки одноопераднда форма команди, тому при відсутності в програмі мовою Асемблера директиви .386 MASM сприймає лише одноопераднді команди IMUL.

Застосування: Команду з трьома операндами доцільно застосовувати для визначення адрес структур у масиві структур, наприклад:

```
person      struc
_name       db      30 dup (?)
grup        db      6 dup (?)
zalic       db      ?
personends

base_person  person 150 dup (<>)

Mov    cx,150
m12:
Imul   si,cx,size person
Mov    byte ptr base_person[si-size person].zalic, '+'
Loop   m12
```

Таблиця Д.25 – Команда **IN**

IN	**Введення з порту**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
Код операції	***Структура команди***	***Приклад на асемблері***					***Такти***			
E4 ib	**IN AL, imm8**	**In al,60h**					**7, pm=4/ 21,vm=19**			
E5 ib	**IN AX, imm8**	**In ax,00h**					**7, pm=4/ 21,vm=19**			
E5 ib	**IN EAX, imm8**	**In eax,81h**					**7, pm=4/ 21,vm=19**			
EC	**IN AL, DX**	**In al,dx**					**7, pm=4/ 21,vm=19**			
ED	**IN AX, DX**	**In ax,dx**					**7,pm=4/ 21,vm=19**			
ED	**IN EAX, DX**	**In eax,dx**					**7,pm=4/ 25,vm=19**			

Схема команди:	IN *акумулятор,адреса порту*

Призначення: введення даних з порту введення-виведення.

Алгоритм роботи: Переслати байт, слово або подвійне слово з порту введення-виведення в акумулятор (регістр al/ax/eax). Адреса порту задається другим операндом як безпосереднє значення або як значення в регістрі dx. Безпосереднім значенням можна задати порти з адресами у діапазоні 0-0ffh. За допомогою регістра dx задаються порти з адресами у діапазоні 0-0ffffh. Розрядність порту визначається розмірністю першого операнда і може бути байтом, словом, або подвійним словом. Для 16-розрядних портів адреси повинні бути кратними 2, для 32-розрядних портів – кратними 4.

Застосування: Команда застосовується для безпосередньої роботи із зовнішніми пристроями комп'ютера за допомогою портів. Як приклад застосування, розглянемо фрагмент обробки переривання від клавіатури. Це переривання викликається кожен раз при натисканні будь-якої клавіші клавіатури. Процедура обробки цього переривання повинна прочитати скан-код клавіші (порт за адресою 60h) та підтвердити клавіатурі факт прийому скан-коду (надіслати імпульс на старший біт порту 61h):

```
;читаємо скан-код
In      al,60h
mov     can_cod,al
;читаємо порт 61h
In      al,61h
;тимчасово збережемо його
Push    ax
;старший біт байту з порту 61h в 1
Or      al,80h
;підтверджуємо факт прийому скан-коду
Out     61h,al
Pop     ax
;завершення імпульсу (відновили байт у порту 61h)
Out     61h,al
```

Таблиця Д.26 – Команда INC

INC	**Інкремент**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
		*				*	*	*	*	
Код операції	***Структура команди***	***Приклад на асемблері***						***Такти***		
FE /0	**INC r/m8**	**Inc al**						**1/3**		
FF /0	**INC r/m16**	**Inc word ptr [di]**						**1/3**		
FF /0	**INC r/m32**	**Inc dword ptr [ebx]**						**1/3**		
40+rw	**INC r16**	**Inc dx**						**1**		
40+rd	**INC r32**	**Inc ecx**						**1**		

Схема команди:	INC *операнд*

Призначення: збільшення значення операнда в пам'яті або в регістрі на 1.

Алгоритм роботи: операнд:=операнд+1.

Застосування: Команда INC використовується для збільшення значення байта, слова, подвійного слова в пам'яті або регістрі на одиницю. *Відзначимо, що ознака cf не змінюється.* Це дозволяє використовувати команду INC для збільшення індексу циклу, а ознаку cf застосовувати в інших командах. Наприклад, зсув вліво на 1 розряд багатобайтного числа, заданого в масиві m1:

```
…
M1      db 100 dup (5)
…
        Mov   cx,100
        Mov   bx,0          ; індекс циклу
        clc
@10:
        Rcl   m1[bx},1
        Inc   bx
        Loop  @10

; в результі m1=(0ah,0ah,…,0ah)
```

Таблиця Д.27 – Команда INT

INT **INT3** **INTO**	**Виклик процедури переривання**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
				0	**0**			*	*	

Код операції	***Структура команди***	***Приклад на асемблері***	***Такти***
CD ib	**INT imm8**	**Int 13**	**16**
CC	**INT3 (int 3)**	**Int3**	**13**
CE	**INTO (int 4)**	**Into**	**виконано: 13** **не виконано: 4**

Схема команди:	INT *константа*

Призначення: виклик підпрограми за логічною адресою в таблиці векторів переривань, з номером переривання, заданим операндом команди.

Алгоритм роботи:

1) записати в стек регістр ознак eflags/flags, після чого адресу повернення. При записі адреси повернення спочатку записується вміст сегментного регістра cs, потім вміст покажчика команд eip/ip;

2) скинути в нуль ознаки if та tf;

3) передати управління на програму обробки переривання із зазначеним номером.

Застосування: Як видно із синтаксису, існують три форми цієї команди:

1) Перша форма команди (INT imm8) займає два байти, має код операції 0cdh і дозволяє ініціювати виклик підпрограми обробки переривання з номером вектора в діапазоні 0-255.

2) INT3 – має свій індивідуальний код операції 0cch і займає один байт. Ця обставина робить її дуже зручною для використання в різних програмних налагоджувачах для встановлення точок зупинки шляхом заміни першого байта будь-якої команди. Мікропроцесор, зустрічаючи в

послідовності команд команду з кодом операції 0cch, викликає програму обробки переривання з номером вектора 3, що слугує для зв'язку з налагоджувачем програм.

3) Третя форма INTO має свій індивідуальний код операції 0ceh і займає один байт. Властивість цієї команди – ініціювати виклик підпрограми обробки переривання з номером вектора 4 – визначає варіанти її застосування. Якщо попередня команда програми може в результаті своєї роботи встановити ознаку переповнення of (наприклад, арифметичні команди), то для виявлення й обробки такої ситуації можна використовувати команду INTO.

```
.486
Mov     bx,186
Imul    eax,bx,8
  ;якщо результату не вистачило розмірності операнда1, то of установиться в 1
  ;виправимо ситуацію в оброблювачі переривання 4
Into
```

Таблиця Д.28 – Команда IRET

IRET	**Повернення з процедури обробки переривання**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
		*	*	*	*	*	*	*	*	*

Код операції	***Структура команди***	***Опис***	***Такти***
CF	**IRET**	**Повернення з процедури обробки переривання у реальному режимі**	**8**

Призначення: використовується в процедурах обробки переривання для повернення в перервану програму.

Алгоритм роботи: У реальному режимі команда IRET послідовно читає зі стеку і відновлює вміст наступних регістрів: eip/ip, cs, eflags/flags.

Застосування: Команду IRET необхідно застосовувати для закінчення процедури обробки переривань.

```
my_int1c    proc        ;програма обробки переривання 1Ch
```

```
                …
                Iret
my_int1c        endp
```

Таблиця Д.29 – Команди Jcc

Jcc	**Передача управління за умовою**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**

Код операції	***Структура команди***	***Умови передачі управління***	***Такти***
1	**2**	**3**	**4**
77 cb	**JA rel8**	**Якщо вище (CF=0 та ZF=0)**	**1**
73 cb	**JAE rel8**	**Якщо вище чи дорівнює (CF=0)**	**1**
72 cb	**JB rel8**	**Якщо нижче (CF=1)**	**1**
76 cb	**JBE rel8**	**Якщо нижче чи дорівнює (CF=1 чи ZF=1)**	**1**
72 cb	**JC rel8**	**Якщо перенесення (CF=1)**	**1**
E3 cb	**JCXZ rel8**	**Якщо CX = 0**	**6,5**
E3 cb	**JECXZ rel8**	**Якщо ECX = 0**	**6,5**
74 cb	**JE rel8**	**Якщо дорівнює (ZF = 1)**	**1**
74 cb	**JZ rel8**	**Якщо нуль (ZF = 1)**	**1**
7F cb	**JG rel8**	**Якщо більше (ZF=0 і SF=OF)**	**1**
7D cb	**JGE rel8**	**Якщо більше чи дорівнює (SF=OF)**	**1**
7C cb	**JL rel8**	**Якщо менше (SF<>OF)**	**1**
7E cb	**JLE rel8**	**Якщо менше чи дорівнює (ZF=1 чи SF<>OF)**	**1**
76 cb	**JNA rel8**	**Якщо не вище (CF=1 чи ZF=1)**	**1**
72 cb	**JNAE rel8**	**Якщо не вище чи дорівнює (CF=1)**	**1**
73 cb	**JNB rel8**	**Якщо не нижче (CF=0)**	**1**
77 cb	**JNBE rel8**	**Якщо не нижче чи дорівнює (CF=0 чи ZF=0)**	**1**

Продовження табл. Д.29

1	2	3	4
73 cb	**JNC rel8**	**Якщо не перенесення (CF=0)**	**1**
75 cb	**JNE rel8**	**Якщо не дорівнює (ZF=0)**	**1**
7E cb	**JNG rel8**	**Якщо не більше (ZF=1 чи SF<>OF)**	**1**
7C cb	**JNGE rel8**	**Якщо не більше чи дорівнює (SF<>OF)**	**1**
7D cb	**JNL rel8**	**Якщо не менше (SF=OF)**	**1**
7F cb	**JNLE rel8**	**Якщо не менше чи дорівнює (ZF=0 і SF=OF)**	**1**
71 cb	**JNO rel8**	**Якщо не переповнення (OF=0)**	**1**
7B cb	**JNP rel8**	**Якщо непарне (PF=0)**	**1**
79 cb	**JNS rel8**	**Якщо додатнє (SF=0)**	**1**
75 cb	**JNZ rel8**	**Якщо не нуль (ZF=0)**	**1**
70 cb	**JO rel8**	**Якщо переповнення (OF=1)**	**1**
7A cb	**JP rel8**	**Якщо парне (PF=1)**	**1**
7A cb	**JPE rel8**	**Якщо парне (PF=1)**	**1**
7B cb	**JPO rel8**	**Якщо непарне (PF=0)**	**1**
78 cb	**JS rel8**	**Якщо від'ємне (SF=1)**	**1**
OF 87 cw	**JA rel16**	**Якщо вище (CF=0 та ZF=0)**	**1**
OF 83 cw	**JAE rel16**	**Якщо вище чи дорівнює (CF=0)**	**1**
OF 82 cw	**JB rel16**	**Якщо нижче (CF=1)**	**1**
OF 86 cw	**JBE rel16**	**Якщо нижче чи дорівнює (CF=1 чи ZF=1)**	**1**
OF 82 cw	**JC rel16**	**Якщо перенесення (CF=1)**	**1**
OF 84 cw	**JE rel16**	**Якщо дорівнює (ZF = 1)**	**1**
OF 84 cw	**JZ rel16**	**Якщо нуль (ZF = 1)**	**1**
OF 8F cw	**JG rel16**	**Якщо більше (ZF=0 і SF=OF)**	**1**
OF 8D cw	**JGE rel16**	**Якщо більше чи дорівнює (SF=OF)**	**1**
OF 8C cw	**JL rel16**	**Якщо менше (SF<>OF)**	**1**
OF 8E cw	**JLE rel16**	**Якщо менше чи дорівнює (ZF=1 чи SF<>OF)**	**1**
OF 86 cw	**JNA rel16**	**Якщо не вище (CF=1 чи ZF=1)**	**1**

Продовження табл. Д.29

1	2	3	4
OF 82 cw	**JNAE rel16**	**Якщо не вище чи дорівнює (CF=1)**	**1**
OF 83 cw	**JNB rel16**	**Якщо не нижче (CF=0)**	**1**

Схема команди:	Jcc *мітка* JCXZ *мітка* JECXZ *мітка*

Призначення: внутрішньосегментна передача управління в залежності від виконання умови.

Алгоритм роботи (крім команд JCXZ/JECXZ):

Умова задається кодом операції (мнемокодом мовою Асемблера), а її виконання залежить від стану однієї або декількох ознак:

1) якщо стан ознак відповідає умові, що перевіряється, то *ip:=ip+rel8* або *ip=ip+rel16;*

2) якщо стан ознак не відповідає умові, що перевіряється, то буде виконуватись наступна команда.

Алгоритм роботи команди JCXZ/JECXZ. Перевірка умови, що вміст регістра ecx/cx дорівнює нулю:

1) якщо вміст ecx/cx дорівнює 0, то *ip:=ip+rel8*;

2) якщо вміст ecx/cx не дорівнює 0, то буде виконуватись наступна команда.

Команда JCXZ використовує регістр cx, команда JECXZ використовує регістр ecx. Код команди для цих команд однаковий, тому транслятор аналізує розрядність даних, яка задана для програми в цілому, і генерує префікс зміни розрядності даних, якщо ця розрядність не співпадає з розрядністю регістра, яка задана мнемокодом.

В командах передачі управління за умовою реалізована відносна адресація, тобто в адресних частинах команд передачі управління задається зміщення цільової команди відносно поточного значення лічильника команд. В схемі команди відносна адреса позначена як *rel8* – однобайтна або *rel16* – двохбайтна. У випадку *rel8* цільова команда повинна розміщуватись на відстані не далі ніж +127 або –128 від поточної адреси. У випадку *rel16,* цільова команда може бути розташована на відстані від –32768 до +32767 відносно поточної адреси. Поточною адресою є адреса наступної за розташуванням в пам’яті команди після команди передачі управління.

Команди JCXZ / JECXZ з rel16 відсутні.

Застосування (крім JCXZ/JECXZ*):* Команди умовного переходу зручно застосовувати для перевірки різних умов, що виникають під час виконання програми. Як відомо, значна частина команд формує ознаки результатів своєї роботи в регістрі eflags/flags. Ця обставина і використовується командами умовного переходу. Вище наведено перелік команд умовного переходу, аналізовані ними ознаки та відповідні логічні умови переходу. Логічні умови "більше" і "менше" відносяться до порівнянь цілих чисел зі знаком, а "вище" і "нижче" – до порівнянь цілих чисел без знаку. Деякі з наведених команд мають різні мнемокоди, але однакові умови переходу. В дійсності, це одна й та ж сама машинна команда. Різні мнемокоди використовуються для підвищення наглядності програм мовою Асемблера. Наприклад, існують дві мнемоніки – jz та je для однієї і тієї ж команди передачі управління, якщо zf=1. Мнемоніку je (передача управління, коли дорівнює) доцільно використовувати після команд порівняння (cmp). Мнемоніку jz (передача управління, коли 0) в інших випадках, наприклад, після команд test.

Для реалізації міжсегментних переходів необхідно комбінувати команди умовного переходу і команду безумовного переходу jmp. При цьому, можна скористатися тим, що практично для будь-якої команди умовного переходу існує команда з протилежною умовою.

Застосування JCXZ/JECXZ*:* Регістр CX/ECX в командах LOOP, LOOPZ, LOOPNZ та в префіксах REP, REPZ, REPNZ використовується як лічильник циклів. При виконанні команд LOOP, LOOPZ, LOOPNZ спочатку від вмісту регістру CX/ECX віднімається 1 і тільки після цього аналізується його вміст на 0. Якщо перед виконанням команди LOOP, LOOPZ, LOOPNZ вміст регістру CX/ECX дорівнює 0, то це означає, що цикл буде виконуватись 2^{16} або 2^{32} раз. Використання команди JCXZ/JECXZ перед циклом дозволяє запобігти цьому явищу, якщо воно не бажане.

З іншої сторони, при використанні команд LOOPZ, LOOPNZ та префіксів REPZ, REPNZ цикл може закінчитись і в випадку, коли CX/ECX не дорівнює 0. Для аналізу цієї обставини після виходу із циклу як раз і доцільно використати команду JCXZ/JECXZ:

```
Jcxz   m1      ;обійти цикл, якщо cx=0
cycl:
       ;деякий цикл
Inc    si
Mov    Ar1[si],al
…
Loop   cycl
m1:
```

Таблиця Д.30 – Команда JMP

JMP	**Безумовний перехід**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
Код операції	***Структура команди***	***Опис типу переходу***						***Такти***		
EB cb	**JMP rel8**	**Короткий відносний, зміщення вказується відносно адреси наступної команди Jmp short label2**						**1**		
E9 cw	**JMP rel16**	**Внутрішньосегментний (ближній) відносний, зміщення вказується відносно адреси наступної команди. Jmp near ptr label2**						**1**		
FF /4	**JMP r/m16**	**Внутрішньосегментний (ближній) посередній, або внутрішньосегментний (ближній) посередній регістровий Jmp word ptr M2[si] або Jmp si**						**2**		
EA cd	**JMP ptr16:16**	**Міжсегментний прямий, вказується 4 байтна логічна адреса Jmp far ptr label2**						**3**		
FF /5	**JMP m16:16**	**Міжсегментний посередній Jmp dword ptr Ar2[bx]**						**4**		

Призначення: використовується в програмі для організації безумовного переходу як в поточному сегменті команд, так і за його межі.

Алгоритм роботи в реальному режимі:

1) якщо внутрішньосегментний перехід, то
   a. якщо коротка відносна адресація (код операції 0EBh), то IP:=IP+(знакове розширення rel8 до 16 розрядів);
   b. якщо відносна адресація (код операції 0E9h), то IP:=IP+rel16;
   c. якщо посередня регістрова адресація (код операції FF/4, mod=11), то IP:=r16;

d. якщо посередня адресація (код операції FF/4, mod ≠ 11), то IP:=m16;

2) якщо міжсегментний перехід, то

a. якщо пряма адресація (код операції 0EAh), то CS:IP:=ptr16:16;

b. якщо посередня адресація (код операції FF /3, mod ≠ 11), то CS:IP:=m32.

Таблиця Д.31 – Команда LAHF

LAHF	**Завантажити ознаки в регістр AH**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**

Код операції	***Структура команди***	***Приклад на асемблері***	***Такти***
9F	**LAHF**	**Lahf**	**2**

Призначення: переслати вміст молодшого байта регістра eflags/flags, в якому містяться п'ять ознак (cf, pf, af, zf і sf) в регістр ah.

*Застосування:*Через те, що регістр ознак безпосередньо недоступний, команду LAHF можна застосовувати для аналізу і наступної зміни командою SAHF стану ознак cf, pf, af, zf і sf регістра eflags/flags.

```
;скинути в нуль ознаку pf
Lahf
And     ah,11111011b
Sahf
```

Таблиця Д.32 – Команди LDS-LSS

LDS **LES** **LFS** **LGS** **LSS**	**Завантажити логічну адресу**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**

Код операції	***Структура команди***	***Приклад на асемблері***	***Такти***
1	**2**	**3**	**4**
C5 /r	**LDS r16, m16:16**	**Lds si, dword ptrX**	**4**
C5 /r	**LDS r32, m16:32**	**Lds esi, pword ptr Z**	**4**

Продовження табл. Д.32

1	2	3	4
0F B2 /r	**LSS r16, m16:16**	**Lss sp, vector**	**4, pm=8**
0F B2 /r	**LSS r32, m16:32**	**Lss esp, pword ptr Bag**	**4, pm=8**
C4 /r	**LES r16, m16:16**	**Les di, dword ptr X**	**4**
C4 /r	**LES r32, m16:32**	**Les edi, pword ptr Z**	**4**
0F B4 /r	**LFS r16, m16:16**	**Lfs ax, dword ptrX**	**4**
0F B4 /r	**LFS r32, m16:32**	**Lfs eax, pword ptr Z**	**4**
0F B5 /r	**LGS r16, m16:16**	**Lgs dx, dword ptrX**	**4**
0F B5 /r	**LGS r32, m16:32**	**Lgs eax, pword ptr Z**	**4**

Схема команди:	LDS *приймач,джерело*
	LES *приймач,джерело*
	LFS *приймач,джерело*
	LGS *приймач,джерело*
	LSS *приймач,джерело*

Призначення: завантаження логічної адреси.

Алгоритм роботи: Алгоритм роботи команди залежить від типу операнда *джерело:*

1) якщо тип 4 (dword ptr), то завантажити перші два байти з комірок пам'яті за адресою *джерело* в 16-розрядний регістр, зазначений операндом *приймач*. Наступні два байти в області *джерело* повинні містити сегментну складову логічної адреси. Вони завантажуються в регістр ds/es/fs/gs/ss;

2) якщо тип 6 (pword ptr), то завантажити перші чотири байти з комірки пам'яті за адресою *джерело* в 32-розрядний регістр, зазначений операндом *приймач*. Наступні два байти в області *джерело* повинні містити сегментну складову (старші 16 розрядів фізичної адреси сегмента або селектор дескриптора сегмента) логічної адреси. Ці два байти завантажуються в регістр ds/es/fs/gs/ss.

Застосування: в процесорах i80x86 відсутня посередня адресація даних, тобто в командах обробки даних не можна вказати адресу адреси даних. Тому для використання логічних адрес, які формуються або зберігаються в пам'яті, застосовують команди LES, LDS, LSS, LGS, LFS.

Типовим застосуванням команд LES та LDS є завантаження логічних адрес рядків (масивів) перед їх обробкою за допомогою ланцюгових команд. Команду LSS SP (LSS ESP) рекомендується використовувати при зміні логічної адреси стеку, оскільки завантаження регістрів SS та ESP/SP за допомогою окремих команд несе потенційну загрозу виникнення помилок при відлагодженні та виконанні програми.

Таблиця Д.33 – Команда LEA

LEA	**Завантажити ефективну адресу**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**

Код операції	***Структура команди***	***Приклад на асемблері***	***Такти***
8D /r	**LEA r16, ea16**	**Lea ax, driv[si+bx]**	**1**
8D /r	**LEA r32, ea16**	**Lea edx, inv[si]**	**1**
8D /r	**LEA r16, ea32**	**Lea di, [eax+ebx*8]**	**1**
8D /r	**LEA r32, ea32**	**Lea eax, priv[eax+8*ecx]**	**1**

Схема команди:	LEA *приймач*, *джерело*

Призначення: завантаження в операнд *приймач* ефективної адреси (зміщення в сегменті), яка задана операндом *джерело*.

Алгоритм роботи:

1) Якщо 16-розрядні дані та 16-розрядна адресація, то в регістр *приймач* завантажується 16-бітне значення ефективної адреси;

2) Якщо 32-розрядні дані та 16-розрядна адресація, то в молодші 16 розрядів регістра *приймач* завантажується 16-бітне значення ефективної адреси, а в старші 16 розрядів записується 0.
3) Якщо 16-розрядні дані та 32-розрядна адресація, то в регістр *приймач* завантажуються молодші 16 біт значення ефективної адреси;
4) Якщо 32-розрядні дані та 32-розрядна адресація, то в регістр *приймач* завантажується 32-бітне значення ефективної адреси.

Застосування: Основне застосування команди lea – забезпечення адресації елементів складних структур даних, багатовимірних масивів тощо.

```
;завантажити в регістр ax елемент mas[i1,i2,i3]
.data

  mas    dw     10 dup (12 dup (14 dup (? )))
  i1     dw     ?
  i2     dw     ?
  i3     dw     ?
.code

  Imul   bx,i1,12*14
  Imul   ecx,i2,14
  Xor    esi,esi
  Mov    si,i3
  Lea    di,[ecx+esi*2]
  Mov    ax,mas[bx+di]        ;ax:=mas[i1,i2,i3]
```

Таблиця Д.34 – Команди LODS

LODS **LODSB** **LODSW** **LODSD**	**Завантажити рядковий операнд**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**

Код операції	***Структура команди***	***Приклад на асемблері***	***Такти***
AC	**LODS m8**	**Lods byte ptr src**	**2**
AD	**LODS m16**	**Lods fs:word ptr src**	**2**
AD	**LODS m32**	**Lods dword ptr src**	**2**
AC	**LODSB**	**Lodsb**	**2**
AD	**LODSW**	**Lodsw**	**2**
AD	**LODSD**	**Lodsd**	**2**

Схема команди:	LODS *джерело*
	LODSB
	LODSW
	LODSD

Призначення: завантажити елемент рядка (ланцюжка, масиву) у регістр al/ax/eax.

Алгоритм роботи:

1) завантажити елемент із комірки пам'яті з логічною адресою, яка вказується в ds:esi/si, у регістр al/ax/eax. Розмір елемента визначається типом операнда *джерело* (для команди lods) або мнемокодом команди (для команд LODSB, LODSW, LODSD);
2) змінити значення регістра si/esi на величину, що дорівнює довжині елемента ланцюжка:
   a. якщо df=0, то збільшити вміст регістра на довжину елемента послідовності;
   b. якщо df=1, то зменшити вміст регістра на довжину елемента послідовності.

Застосування: Після виконання порівняння двох масивів зручно використати команду LODS для завантаження в регістр al/ax/eax елемента масиву, який не співпав з елементом еталонного масиву (див. приклад для команди CMPS). Перед командою LODS можна вказати префікс повторення REP, але в цьому немає особливого сенсу.

Таблиця Д.35 – Команда LOOP

LOOP **LOOPE** **LOOPZ** **LOOPNE** **LOOPNZ**	**Організація циклу**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**

Код операції	***Структура команди***	***Приклад на асемблері***	***Такти***
E2 ib	**Loop rell8**	**Loop @10**	**1**
E1 ib	**Loope rell8**	**Loope @10**	**2**
E1 ib	**Loopz rell8)**	**Loopz @10**	***2***
E0 ib	**Loopne rell8**	**Loopne @10**	**2**
E0 ib	**Loopnz rell8**	**Loopnz @10**	**2**

Призначення: організація циклів в програмах.

Алгоритм роботи:

1) CX=CX-1
2) Якщо команда LOOP (код операції E2) і CX=0, то IP=IP+ rell8
3) Якщо команда LOOPE або LOOPZ (код операції E1) і CX=0 та Zf=1, то IP=IP+ rell8
4) Якщо команда LOOPNE або LOOPNZ (код операції E0) і CX=0 та Zf=0, то IP=IP+ rell8

Застосування: може використовуватись для реалізації операторів циклів мов високого рівня.

Недоліком команди є первопочатковий декремент CX, що в окремих випадках може призвести до негативних результатів. Див. команду JCXZ.

Таблиця Д.36 – Команда MOV

MOV	**Переслати дані**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**

Код операції	***Структура команди***	***Приклад на асемблері***	***Такти***
88 /r	**MOV r/m8, r8**	**Mov [di], dl**	**1**
89 /r	**MOV r/m16, r16**	**Mov [si], cx**	**1**
89 /r	**MOV r/m32, r32**	**Mov [ebx], edi**	**1**
8A /r	**MOV r8, r/m8**	**Mov dh, cl**	**1**
8B /r	**MOV r16, r/m16**	**Mov dx,[di]**	**1**
8B /r	**MOV r32, r/m32**	**Mov ebx, [edi]**	**1**
8C /r	**MOV r/m16, Sreg**	**Mov dx, ds**	**1**
8E /r	**MOV Sreg, r/m16**	**Mov ds, si**	**2/3**
A0	**MOV AL, moffs8**	**Mov al, byteC**	**1**
A1	**MOV AX, moffs16**	**Mov ax, wordR**	**1**
A1	**MOV EAX, moffs32**	**Mov eax, dwd**	**1**
A2	**MOV moffs8, AL**	**Mov byteC, al**	**1**
A3	**MOV moffs16, AX**	**Mov wordR, ax**	**1**
A3	**MOV moffs32, EAX**	**Mov dwd, eax**	**1**
B0+rb	**MOV reg8, imm8**	**Mov dl, 9**	**1**
B8+rw	**MOV reg16, imm16**	**Mov cx, 3212**	**1**
B8+rd	**MOV reg32, imm32**	**Mov ebp, 21367**	**1**
C6 /0	**MOV r/m8, imm8**	**Mov [di], 2**	**1**
C7 /0	**MOV r/m16, imm16**	**Mov [di], 2354**	**1**
C7 /0	**MOV r/m32, imm32**	**Mov [esi], 985**	**1**

Схема команди:	MOV *приймач,джерело*

Призначення: пересилання даних між регістрами або між регістром і пам'яттю.

Алгоритм роботи: джерело:=приймач.

Застосування: Команда MOV застосовується для різного роду пересилань даних, при цьому, незважаючи на всю простоту цієї дії, необхідно пам'ятати про деякі обмеження й особливості виконання операції:

1) операнд *джерело* може бути адресою в пам’яті, регістром або безпосередніми даними;
2) операнд *приймач* може бути адресою в пам’яті або регістром;
3) значення операнда *джерело* не змінюється;
4) обидва операнди одночасно не можуть бути адресою пам'яті (при необхідності можна використовувати ланцюгову команду movs);
5) якщо *приймач* є сегментним регістром, то операнд *джерело* не може бути безпосередніми даними;
6) сегментний регістр CS не може бути операндом *приймач;*
7) команди з кодами від A0 та A3 в якості одного із операндів мають регістр al/ax/eax, а інший операнд (moffs) – зміщення в команді, яке використовується як зміщення в сегменті; таким чином байт modR/m в цих командах відсутній;
8) в якості безпосередніх даних може використовуватись зміщення в сегменті адресних виразів, при цьому використовується оператор offset, наприклад – Mov bx,offset mas;
9) в якості безпосередніх даних може використовуватись ім’я сегменту.

Таблиця Д.37 – Команди MOVS

MOVS **MOVSB** **MOVSW** **MOVSD**	**Переслати дані з рядка в рядок**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**

Код операції	***Структура команди***	***Приклад на асемблері***	***Такти***
A4	**MOVS m8, m8**	**Movs byte ptr [edi], fs:byte ptr [esi]**	**4**
A5	**MOVS m16, m16**	**Movs word ptr [edi], gs:word ptr [esi]**	**4**
A5	**MOVS m32, m32**	**Movs dword ptr [edi],gs:dword ptr [esi]**	**4**
A4	**MOVSB**	**Movsb**	**4**
A5	**MOVSW**	**Movsw**	**4**
A5	**MOVSD**	**Movsd**	**4**

Схема команди:	MOVS *приймач,джерело* MOVSB MOVSW MOVSD

Призначення: пересилання елементів рядків (послідовностей, ланцюжків, масивів) у пам'яті.

Алгоритм роботи:

1) виконати копіювання байта, слова або подвійного слова з операнда *джерело* в операнд *приймач*, при цьому логічні адреси елементів попередньо повинні бути завантажені:
   a. адреса *джерела* – у пари регістрів ds:esi/si (дозволяється використання префікса заміни сегменту);
   b. адреса *приймача* – у пари регістрів es:edi/di (заміна сегмента не допускається);
2) в залежності від стану ознаки df змінити значення регістрів esi/si та edi/di:
   a. якщо df=0, то збільшити вміст цих регістрів на довжину елемента послідовності (пересилання розпочинається з початку рядка);
   b. якщо df=1, то зменшити вміст цих регістрів на довжину елемента послідовності (пересилання розпочинається з кінця рядка);
   c. якщо є префікс повторення, то виконати обумовлені ним дії (див. команду rep).

При трансляції команд MOVSW та MOVSD Асемблер генерує префікс заміни розрядності даних (код 66h), якщо розрядність даних за замовчуванням не відповідає мнемокоду команди. Розрядність адрес для

команд MOVSB, MOVSW та MOVSD (відповідно використання регістра SI та DI чи ESI та EDI) встановлюється тільки за замовчуванням. При трансляції команди movs Асемблер може згенерувати префікс заміни розрядності даних, префікс заміни розрядності адрес і один із префіксів заміни сегменту. Нехай за замовчуванням встановлені 16-розрядні дані та адреси (атрибут USE16 в директиві SEGMENT), тоді в результаті трансляції машинної інструкції

```
Movs        dword ptr [edi],gs:dword ptr [esi]
```

Асемблер згенерує 4 байти:

67h| 66h| 65h: 0A5h, де 67h – префікс заміни розрядності адрес, 66h – префікс заміни розрядності даних, 65h – префікс заміни сегменту, 0A7h – код команди movsw. Таким чином, поле операндів в машинній інструкції movs фактично використовується для генерації префіксів та уточнення коду операції.

ЗАУВАЖЕННЯ. В реальному режимі при використанні регістрів ESI та EDI старші 16 розрядів 32-розрядного зміщення обнуляються.

Застосування: Команду MOVS (MOVSB, MOVSW, MOVSD) застосовують для пересилання рядків (послідовностей, ланцюжків, масивів) у пам'яті. Для цього необхідно використовувати префікс rep. Префікс REP змушує циклічно виконувати команди пересилання доти, поки вміст регістра ecx/cx не дорівнюватиме нулю. При накладанні рядків *приймача* та *джерела* необхідно визначати порядок пересилання – з початку чи з кінця рядка.

```
str1        db      'str1 копіюється в str2'
len_str1=$-str1
a_str1 dd   str1
str2        db      len_str1 dup (' ')
a_str2 dd   str2

Mov         cx,len_str1
Lds         si,str1
```

```
Les     di,str2
Cld
Rep     Movsb
```

Таблиця Д.38 – Команда MOVSX

MOVSX	**Пересилання з розширенням знаку**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**

Код операції	***Структура команди***	***Приклад на асемблері***	***Такти***
0F BE /r	**MOVSX r16, r/m8**	**Movsx ax, cl**	**3**
0F BE /r	**MOVSX r32, r/m8**	**Movsx eax, byte ptr memo**	**3**
0F BE /r	**MOVSX r32, r/m16**	**Movsx ecx, word ptr [edi]**	**3**

Схема команди:	MOVSX *приймач,джерело*

Призначення: перетворення елементів зі знаком меншої розмірності в еквівалентні їм елементи зі знаком більшої розмірності.

Алгоритм роботи:

1) записати вміст операнда *джерела* в операнд *приймач*, починаючи з молодших розрядів *джерела*;

2) поширити значення знакового розряду *джерела* на вільні старші розряди операнда призначення.

Застосування: Команду MOVSX використовують для одержання еквівалентного, але більшого за розміром операнда зі знаком. Це може знадобитися для приведення розміру (типу) операнда до необхідного для використання в інших командах. Наприклад, необхідно до 4-х байтного цілого додати 2-х байтне ціле:

```
v1      dw      -789
v2      dd      1234568

movsx           eax,v1
add             v2,eax
```

Таблиця Д.39 – Команда MOVZX

MOVZX	**Пересилання з розширенням нулями**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**

Код операції	***Структура команди***	***Приклад на асемблері***	***Такти***
0F B6 /r	**MOVZX r16, r/m8**	**Movzx bx, ah**	**3**
0F B6 /r	**MOVZX r32, r/m8**	**Movzx ebx, cl**	**3**
0F BE /r	**MOVZX r32, r/m16**	**Movzx ecx, word ptr [ebx]**	**3**

Схема команди:	MOVZX *приймач,джерело*

Призначення: перетворення беззнакових елементів меншої розрядності в еквівалентні їм беззнакові елементи більшої розрядності.

Алгоритм роботи:

1) записати вміст операнда *джерело* в операнд *приймач*, починаючи з молодших розрядів;
2) записати нуль на вільні старші розряди операнда *приймач*.

Застосування: Команду MOVZX використовують для одержання еквівалентного, але більшого за розміром операнда без знаку, тобто для узгодження операндів різної розрядності (різного типу). Наприклад, необхідно до 4-х байтного беззнакового цілого додати 2-х байтне беззнакове ціле:

```
v1      dw      0f780h
v2      dd      110000h

Movzxeax,v1          ;eax:=0000f780h
Add    v2,eax;
```

Таблиця Д.40 – Команда MUL

MUL	**Множення беззнакових чисел**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
		*				?	?	?	?	*

Код операції	***Структура команди***	***Приклад на асемблері***	***Такти***
F6 /4	**MUL AL, r/m8**	**Mul ch**	**11**
F7 /4	**MUL AX, r/m16**	**Mul word ptr [bx]**	**11**
F7 /4	**MUL EAX, r/m32**	**Mul edi**	**10**

Схема команди:	MUL *множник_1*

Призначення: множення двох цілих беззнакових чисел.

Алгоритм роботи: Алгоритм залежить від формату операнда команди і вимагає явної вказівки місця розташування тільки одного множника, що може бути розташований у пам'яті або регістрі. Місце розташування другого множника фіксоване і залежить від розміру першого множника:

1) якщо операнд, зазначений у команді, байт, тоді другий множник повинен розташовуватися в al;

2) якщо операнд, зазначений у команді, слово, тоді другий множник повинен розташовуватися в ax;

3) якщо операнд, зазначений у команді, подвійне слово, тоді другий множник повинен розташовуватися в eax.

Результат множення міститься також у фіксованому місці, обумовленому розміром співмножників:

1) при множенні байтів результат міститься в ax;

2) при множенні слів результат міститься в парі dx:ax;

3) при множенні подвійних слів результат міститься в парі edx:eax;

4) якщо старша половина результату нульова, то OF:=0, CF:=0;

5) якщо старша половина результату не нульова, то OF:=1, CF:=1.

Застосування:

```
mn_1   db     115
mn_2   db     250

Mov    al,mn_1
Mul    mn_2          ;ax:=115*250, cf:=1, of:=1
```

Таблиця Д.41 – Команда NEG

NEG	**Зміна знаку**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
		*				*	*	*	*	*

Код операції	***Структура команди***	***Приклад на асемблері***	***Такти***
F6 /3	**NEG r/m8**	**neg byte ptr [bx+6]**	**1/3**
F7 /3	**NEG r/m16**	**neg cx**	**1/3**
F7 /3	**NEG r/m32**	**neg eax**	**1/3**

Схема команди:	NEG *джерело*

Призначення: зміна знаку у доповняльному коді.

Алгоритм роботи:

1) виконати віднімання (0 - *джерело*) і помістити результат на місце операнда *джерело*;

2) якщо *джерело*=0, то cf=0, в інших випадках cf=1.

Застосування: Якщо операнд – додатнє число, то команда neg формує від’ємне число у доповняльному коді. Якщо операнд – від’ємне число у доповняльному коді, то команда neg формує модуль від’ємного числа. Команду neg доцільно використовувати для формування результату віднімання від константи. Наприклад, необхідно від константи 0ff0fh відняти вміст регістру ax (команди sub *константа*,ax не існує):

```
Sub   ax,0ff0fh
Neg   ax          ;ax:=0 - (ax - 0ff0h) = 0ff0fh-ax
```

Таблиця Д.42 – Команда NOP

NOP	**Немає операції**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**

Код операції	***Структура команди***	***Приклад на асемблері***	***Такти***
90	**NOP**	**Nop**	**1**

Схема команди:	NOP

Призначення: порожня команда.

Алгоритм роботи: не виконувати жодних дій (за виключенням того, що (E)IP=(E)IP+1).

Застосування: Команда NOP може використовуватися для резервування місця в сегменті кодів без зміни алгоритму програми, або для видалення шляхом заміни на команду nop окремих команд при відналагоджуванні програм.

Таблиця Д.43 – Команда NOT

NOT	**Порозрядне логічне “НІ”**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**

Код операції	***Структура команди***	***Приклад на асемблері***	***Такти***
F6 /2	**NOT r/m8**	**Not ah**	**1/3**
F7 /2	**NOT r/m16**	**Not word ptr [bp+5]**	**1/3**
F7 /2	**NOT r/m32**	**Not edi**	**1/3**

Схема команди:	NOT *джерело*

Призначення: інвертування всіх бітів операнда *джерело*.

Алгоритм роботи: інвертувати всі біти операнда джерела: з 1 у 0, з 0 у 1, або *джерело*:=2^n – 1 – *джерело*, де n – розрядність операнда *джерело*.

Застосування: Команда not застосовується в різноманітних порозрядних логічних перетвореннях при обробці бітових полів. Вона також може застосовуватись для формування оберненого коду операнда *джерело*.

Таблиця Д.44 – Команда OR

OR	**Логічне “АБО”**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
		0				*****	*****	**?**	*****	**0**

Код операції	***Структура команди***	***Приклад на асемблері***	***Такти***
0C ib	**OR AL, imm8**	**Or al, 0AAh**	**1**
0D iw	**OR AX, imm16**	**Or ax, 0FODh**	**1**
0D id	**OR EAX, imm32**	**Or eax, 23456789h**	**1**
80 /1 ib	**OR r/m8, imm8**	**Or byte ptr [di], 5**	**1/3**
81 /1 iw	**OR r/m16, imm16**	**Or dx,0DBBh**	**1/3**
81 /1 id	**OR r/m32, imm32**	**Or edx,0CAAAAh**	**1/3**
83 /1 ib	**OR r/m16, imm8**	**Or cx, 0Ah**	**1/3**
83 /1 ib	**OR r/m32, imm8**	**Or ecx, 02h**	**1/3**
08 /r	**OR r/m8, r8**	**Or [di], ah**	**1/3**
09 /r	**OR r/m16, r16**	**Or bx, si**	**1/3**
09 /r	**OR r/m32, r32**	**Or memory, eax**	**1/3**
0A /r	**OR r8, r/m8**	**Or dl,sum**	**1/2**
0B /r	**OR r16, r/m16**	**Or di, [si+12]**	**1/2**
0B /r	**OR r32, r/m32**	**Or ecx,raznost**	**1/2**

Схема команди:	OR *приймач, джерело*

Призначення: виконання порозрядної логічної операції “Або”.

Алгоритм роботи:

1) кожен біт результату дорівнює 0, якщо відповідні біти обох операндів дорівнюють 0, в інших випадках біт результату дорівнює 1;
2) записати результат операції в *приймач*;

ЗАУВАЖЕННЯ. Співвідношення розміру операндів – згідно зауваження до команди adc.

Застосування: команду зручно використовувати для примусового встановлення в 1 будь-якої сукупності розрядів *приймача* без зміни вмісту решти розрядів, наприклад:

; нехай регістр dx містить значення 0000011001000010b=0642h

Or dx,1111000000001111 ; dx=1111011001001111b=0f64fh

Таблиця Д.45 – Команда OUT

OUT	**Виведення в порт**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
Код операції	***Структура команди***	***Приклад на асемблері***				***Такти***				
E6 ib	**OUT imm8, AL**	**Out 60h, al**				**12**				
E7 ib	**OUT imm8, AX**	**Out 00h, ax**				**12**				
E7 ib	**OUT imm8, EAX**	**Out 81h, eax**				**12**				
EE	**OUT DX, AL**	**Out dx, al**				**12**				
EF	**OUT DX, AX**	**Out dx, ax**				**12**				
EF	**OUT DX, EAX**	**Out dx, eax**				**12**				

Схема команди:	OUT *адреса порту,акумулятор*

Призначення: виведення значення в порт введення-виведення.

Алгоритм роботи: передати байт, слово або подвійне слово з регістра al/ax/eax у порт, адреса якого визначається першим операндом.

Застосування: Команда застосовується для прямого управління зовнішніми пристроями комп'ютера за допомогою портів. Адреса порту задається першим операндом у безпосередньому вигляді або значенням регістру dx. Безпосереднім значенням можна задати порти з адресами у діапазоні 0-0ffh. За допомогою регістра dx задаються порти з адресами у діапазоні 0-0ffffh. Розрядність порту визначається розмірністю другого операнда і може бути байтом, словом, або подвійним словом. Для 16-розрядних портів адреси повинні бути кратні 2, для 32-розрядних портів – кратні 4. Див. приклад для команди IN.

Таблиця Д.46 – Команда POP

POP	**Прочитати зі стеку**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**

Код операції	***Структура команди***	***Приклад на асемблері***	***Такти***
BF /0	**POP m16**	**Pop word ptr [di]**	**3**
BF /0	**POP m32**	**Pop dword ptr mem**	**3**
58+rw	**POP r16**	**Pop cx**	**1**
58+rd	**POP r32**	**Pop edi**	**1**
1F	**POP DS**	**Pop ds**	**3**
07	**POP ES**	**Pop es**	**3**
17	**POP SS**	**Pop ss**	**3**
0F AL	**POP FS**	**Pop fs**	**3**
0F A9	**POP GS**	**Pop gs**	**3**

Схема команди:	POP *приймач*

Призначення: читання (виштовхування) слова або подвійного слова зі стеку.

Алгоритм роботи:

1) завантажити в *приймач* 2 або 4 байти, логічна адреса яких знаходиться в парі регістрів ss:esp/sp;
2) збільшити вміст esp/sp на 2 або 4.

Розрядність зміщення в сегменті стеку, відповідно використання регістра esp або sp, визначається за замовчуванням. У реальному режимі завжди використовується sp. Ані TASM, ані MASM для команди pop автоматично не генерують префікс зміни розрядності адрес, але при необхідності можуть згенерувати префікс зміни розрядності даних, якщо розрядність операнда *приймач* не співпадає з розрядністю, заданою директивою Segment

Операндом *приймач* можуть бути 16 або 32-розрядні регістри даних, сегментні регістри (за виключенням CS) та адреси (зміщення в сегменті) пам’яті.

Застосування: Команда застосовується для завантаження регістра даних, сегментного регістра або комірок пам'яті вмістом верхівки стеку. Відмітимо, що команди pop cs не існує. Після виконання команди pop ss на період виконання наступної команди блокуються будь-які переривання, включаючи NMI (переривання, яке не маскується). Наступною командою повинна бути команда завантаження регістра esp/sp. Взагалі, завантаження пари ss:esp/sp рекомендується виконувати за допомогою команди LSS SP.

```
my_proc proc near
Push   ax
Push   bx
       ;тіло процедури, у якій змінюється вміст
       ;регістрів ax і bx

Pop    bx
Pop    ax
Ret
endp
```

Таблиця Д.47 – Команди POPA, POPAD

POPA **POPAD**	**Прочитати зі стеку вміст усіх регістрів даних**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**

Код операції	***Структура команди***	***Приклад на асемблері***	***Такти***
61	**POPA**	**Popa**	**5**
61	**POPAD**	**Popad**	**5**

Призначення: завантаження зі стеку всіх регістрів даних в послідовності di/edi, si/esi, bp/ebp, sp/esp, bx/ebx, dx/edx, cx/ecx, ax/eax.

Алгоритм роботи:

1) для команди popa:

```
pop di; pop si; pop bp;
add sp,2;
```

pop bx; pop dx; pop cx; pop ax

2) для команди popad:

pop edi; pop esi; pop ebp;

add sp,4;

pop ebx; pop edx; pop ecx; pop eax

Застосування: Команда POPA за принципом роботи є протилежною команді PUSHA і використовується для відновлення вмісту всіх регістрів даних значеннями зі стеку. Цю команду зручно використовувати по завершенню процедур, особливо процедур обробки переривань, для відновлення регістрів загального призначення перерваної програми. З точки зору оптимізації програм за швидкодією, застосування цієї команди є проблематичним за умов, коли немає необхідності зберігати вміст більшої частини регістрів.

```
.386
    my_proc proc near
    Pusha
                        ;тіло процедури, у якій змінюється
                        ;вміст регістрів загального призначення

    Popa
    ;значення в регістрі sp те ж саме, що й до виконання команди pusha
    Ret
    endp
```

Таблиця Д.47 – Команди POPF, POPFD

POPF **POPFD**	Читання зі стеку в регістр ознак	**O** *****	**D** *****	**I** *****	**T** *****	**S** *****	**Z** *****	**A** *****	**P** *****	**C** *****
Код операції	***Структура команди***	***Приклад на асемблері***						***Такти***		
9D	**POPF**	**Popf**						**6, pm=4**		
9D	**POPFD**	**Popfd**						**6, pm=4**		

Призначення: завантаження зі стеку регістра ознак flags/eflags.

Алгоритм роботи:

1) якщо команда POPF, то

a. завантажити регістр flags словом із верхівки стеку

b. збільшити значення покажчика стеку sp на 2.

2) якщо команда POPFD, то

a. завантажити регістр eflags подвійним словом із верхівки стеку

b. збільшити значення покажчика стеку sp на 4.

ЗАУВАЖЕННЯ. Транслятор TASM автоматично генерує префікс зміни розрядності даних (66h), якщо заданий атрибут розрядності сегмента кодів не відповідає мнемокоду. В той же час, автоматичне генерування префікса зміни розрядності адрес (тобто використання регістра sp чи esp) не передбачено.

Застосування: Команда POPF за принципом роботи є протилежною команді PUSHF і використовується для відновлення зі стеку вмісту регістра ознак. Можливим варіантом використання цієї команди є програми, у яких необхідно зберігати деякий локальний контекст процесу обчислення. Крім того, в системі команд процессорів 80x86 відсутні команди для примусового встановлення значень цілого ряду ознак (наприклад, pf, zf, sf, tf, iopl та ін.). Тому команду POPF/POPFD застосовують для примусового встановлення ознак, попередньо записавши в стек необхідне слово чи подвійне слово. наприклад, реалізувати вираз

; if (ax=bx) or (si=di) or (cx<>dx) then

```
Push   ax
Mov    bp,sp
Cmp    ax,bx
Pushf
Cmp    si,di
Pushf
Cmp    cx,dx
Pushf
Pop    ax
Or     ax,[bp-2]
Not    word ptr [bx-4]
Or     ax.[bp-4]
Popf
Popf
Popf
Push   ax
```

```
  Popf
  Pop    ax
  Jnz …
;приклад 2 - встановити значення регістра flags у 03h
  Mov    ax,3h
  Push   ax
  Popf
```

Таблиця Д.48 – Команда PUSH

PUSH	**Помістити операнд в стек**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**

Код операції	***Структура команди***	***Приклад на асемблері***	***Такти***
FF /6	**PUSH r/m16**	**Push word ptr [si]**	**1/2**
FF /6	**PUSH r/m32**	**Push dword ptr {eax]**	**1/2**
50+rw	**PUSH r16**	**Push di**	**1**
50+rd	**PUSH r32**	**Push esi**	**1**
6A	**PUSH imm8**	**Push 0Ah**	**1**
68	**PUSH imm16**	**Push 023Dh**	**1**
68	**PUSH imm32**	**Push 012345FFh**	**1**
0E	**PUSH CS**	**Push cs**	**1**
16	**PUSH SS**	**Push ss**	**1**
1E	**PUSH DS**	**Push ds**	**1**
06	**PUSH ES**	**Push es**	**1**
0F A0	**PUSH FS**	**Push fs**	**1**
0F A8	**PUSH GS**	**Push gs**	**1**

Схема команди:	PUSH *джерело*

Призначення: запис (заштовхування) операнда *джерело* в стек.

Алгоритм роботи:

1) зменшити вміст esp/sp на 2 або 4;

2) записати 2 або 4 байти вмісту операнда *джерело* в ОЗП за адресою, яка задана в ss:esp/sp.

Розрядність зміщення в сегменті стеку, відповідно використання регістра esp або sp, визначається за замовчуванням. У реальному режимі

завжди використовується sp. Ні TASM, ані MASM для команди push автоматично не генерують префікс зміни розрядності адрес, але при необхідності можуть згенерувати префікс зміни розрядності даних, якщо розрядність операнда *джерело* не співпадає з розрядністю, заданою директивою Segment.

Операндом *джерело* можуть бути 16 або 32-розрядні регістри даних, сегментні регістри, адреса пам’яті (зміщення в сегменті), безпосередні дані.

Застосування: Команда PUSH використовується для організації тимчасового зберігання даних у стеку, наприклад, для збереження вмісту регістра при його тимчасовому використанні за іншим призначенням, для фактичних параметрів процедур та ін. Розмір записуваних значень – або слово, або подвійне слово. Також у стек можна записувати безпосередні значення. На відміну від команди POP, у стек можна включати значення сегментного регістра cs. Інший цікавий момент зв'язаний з регістром sp. Команда Push esp/sp записує в стек значення esp/sp за станом до виконання цієї команди. При записі в стек 8-бітних значень для них виділяється слово або подвійне слово (у залежності від діючої розрядності даних).

```
my_proc proc near
Push   ax
Push   bx
       ;тіло процедури, у якій змінюється вміст
       ;регістрів ax і bx

Pop    bx
Pop    ax
Ret
my_proc endp
```

Таблиця Д.49 – Команди PUSHA, PUSHAD

PUSHA **PUSHAD**	**Запис в стек вмісту всіх регістрів даних**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**

Код операції	***Структура команди***	***Приклад на асемблері***	***Такти***
60	**PUSHA**	**Pusha**	**5**
60	**PUSHAD**	**Pushad**	**5**

Призначення: запис в стек регістрів в наступній послідовності: ax/eax, cx/ecx, dx/edx, bx/ebx, sp/esp, bp/ebp, si/esi, di/edi.

Алгоритм роботи:

1) для команди PUSHA:

   temp=sp;

   Push ax, Push cx, Push dx, Push bx;

   Push *temp*;

   Push bp, Push si, Push di;

2) для команди PUSHAD:

   temp=esp,

   Push eax, Push ecx, Push edx, Push ebx;

   Push *temp*;

   Push ebp, Push esi, Push edi.

Застосування: Команда PUSHA використовується разом з командою POPA для збереження і відновлення всіх регістрів. Цю команди зручно використовувати на початку процедур, особливо процедур обробки переривань для збереження частини контексту деякого обчислювального процесу. З точки зору оптимізації програм за швидкодією застосування цієї команди є проблематичним за умов, коли немає необхідності зберігати вміст більшої частини регістрів.

```
my_proc proc near
    pusha
;тіло процедури, у якій змінюється вміст регістрів
```

```
        Popa
;значення в регістрі sp те ж, що і до виконання команди pusha
        Ret

my_proc endp
```

Таблиця Д.50 – Команди PUSHF, PUSHFD

PUSHF **PUSHFD**	**Запис вмісту регістра ознак в стек**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**

Код операції	***Структура команди***	***Приклад на асемблері***	***Такти***
9C	**PUSHF**	**Pushf**	**4, pm=3**
9C	**PUSHFD**	**Pushfd**	**4, pm=3**

Призначення: запис (заштовхування) в стек вмісту регістра ознак flags/eflags.

Алгоритм роботи:

1) зменшити значення покажчика стеку sp на 2 (на 4);
2) записати у вершину стеку вміст регістра flags (eflags).

Якщо мнемокод команди не відповідає встановленій розрядності даних атрибутом розрядності поточного логічного сегменту, то Асемблер автоматично генерує префікс зміни розрядності даних. В Асемблері відсутні засоби автоматичного генерування префікса зміни розрядності адрес. У реальному режимі для адресації верхівки стеку завжди використовується регістр SP.

Застосування: Оскільки значна частина команд може змінити вміст регістру ознак, то команда pushf/pushfd використовується у випадках, коли необхідно тимчасово зберегти, а потім за допомогою команди POPF/POPFD відновити стан ознак. Крім того, команда забезпечує можливість обробки значень будь-яких ознак із регістру ознак, наприклад:

```
Pushfd
Pop    eax
Shl    eax,12
```

And eax,3 ; eax=iopl

Таблиця Д.51 – Команда RCL

RCL	**Циклічний зсув вліво через ознаку перенесення (CF)**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
		*				*				*

Код операції	***Структура команди***	**Приклад на асемблері**	**Такти**
D0 /2	**RCL r/m8, 1**	**Rcl ah, 1**	**1/3**
D2 /2	**RCL r/m8, CL**	**Rcl dh, cl**	**7-24/9-26**
C0 /2 ib	**RCL r/m8, imm8**	**Rcl al, 4**	**8-25/10-27**
D1 /2	**RCL r/m16, 1**	**Rcl bx, 1**	**1/3**
D3 /2	**RCL r/m16, CL**	**Rcl di, cl**	**7-24/9-26**
C1 /2 ib	**RCL r/m16, imm8**	**Rcl di, 9**	**8-25/10-27**
D1 /2	**RCL r/m32, 1**	**Rcl esi, 1**	**1/3**
D3 /2	**RCL r/m32, CL**	**Rcl ecx, 12**	**7-24/9-26**
C1 /2 ib	**RCL r/m32, imm8**	**Rcl edx, 8**	**8-25/10-27**

Схема команди:	RCL *дані, кількість зсувів*

Призначення: виконання операції циклічного зсуву вліво (у бік старших розрядів) через ознаку перенесення cf.

Операнд *дані* може бути іменем 8, 16- або 32-розрядного регістра даних, або адресою пам’яті (зміщення в сегменті). Операнд *кількість* може бути константою або регістром CL.

Алгоритм роботи:

1) зсув всіх бітів вмісту операнда *дані* вліво на один розряд, при цьому старший біт стає значенням ознаки перенесення cf;
2) одночасно старе значення ознаки перенесення cf записується в молодший біт (стає значенням молодшого біту вмісту операнда *дані*);
3) зазначені вище дві дії повторюються (*кількість* ***mod*** 32) разів.

Якщо операнд *дані* – адреса, тоді необхідно забезпечувати значення типу даних (кількість байтів 1, 2 чи 4), які зсуваються. Якщо дані зсуваються в регістрі CL (CX/ECX), то в якості лічильника зсувів використовується першопочатковий вміст CL. Якщо зсув більш ніж на один розряд, то ознака CF встановлюється за значенням останнього зсуву, а ознака OF є не визначеною. В операціях зсуву на один розряд значення старшого (знакового) розряду *після зсуву* дорівнює (OF ***xor*** CF).

Застосування: Команда RCL застосовується при обробці бітових полів, наприклад, вона дозволяє просто реалізувати лінійний зсув багатобайтних даних на один розряд вліво:

```
    m1    db 100 dup (055h)

    Clc                 ;забезпечити 0 в молодший біт
    Mov   si,-100
@2:
    Rcl   m1[si+100],1
                        ;перший раз адреса m1, другий раз m1+1 і т.д.
    Inc   si            ;ознака cf не змінюється (див. команду inc)
    Jnz   @2
                        ; після виконання зсуву в cf міститься значення
                        ; старшого біта старшого байта
```

Таблиця Д.52 – Команда RCR

RCR	**Циклічний зсув вправо через ознаку перенесення (CF)**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
		*								*

Код операції	***Структура команди***	***Приклад на асемблері***	***Такти***
D0 /3	**RCR r/m8, 1**	**Rcr bh, 1**	**1/3**
D2 /3	**RCR r/m8, CL**	**Rcr bh, cl**	**7-24/9-26**
C0 /3 ib	**RCR r/m8, imm8**	**Rcr cl, 4**	**8-25/10-27**
D1 /3	**RCR r/m16, 1**	**Rcr cx, 1**	**1/3**
D3 /3	**RCR r/m16, CL**	**Rcr si, cl**	**7-24/9-26**
C1 /3 ib	**RCR r/m16, imm8**	**Rcr di, 6**	**8-25/10-27**
D1 /3	**RCR r/m32, 1**	**Rcr edi, 1**	**1/3**
D3 /3	**RCR r/m32, CL**	**Rcr ebx, 12**	**7-24/9-26**
C1 /3 ib	**RCR r/m32, imm8**	**Rcr ebp, 8**	**8-25/10-27**

Схема команди:	RCR *дані, кількість зсувів*

Призначення: виконання операції циклічного зсуву вправо (у бік молодших розрядів) через ознаку перенесення cf.

Операнд *дані* може бути іменем 8, 16- або 32-розрядного регістра даних, або адресою пам'яті (зміщення в сегменті). Операнд *кількість* може бути константою або регістром CL.

Алгоритм роботи:

1) зсув всіх бітів операнда *дані* вправо на один розряд, при цьому молодший біт стає значенням ознаки перенесення cf;

2) одночасно старе значення ознаки перенесення cf записується в старший біт (стає значенням старшого біту операнда *дані*);

3) зазначені вище дві дії повторюються (*кількість* ***mod*** 32) разів.

Якщо операнд *дані* – адреса, то необхідно забезпечувати значення типу даних (кількість байтів – 1, 2 чи 4), які зсуваються. Якщо дані зсуваються в регістрі CL (CX/ECX), то в якості лічильника зсувів використовується першопочатковий вміст CL. Якщо зсув більш ніж на один розряд, то ознака CF встановлюється за значенням останнього зсуву, а ознака OF є не визначеною. В операціях зсуву на один розряд значення OF визначається як результат операціїї ***xor*** між старшим (знаковим) розрядом та CF *до зсуву*.

Застосування: Команда RCR застосовується при обробці бітових полів, наприклад, вона дозволяє просто реалізувати як лінійний, так і арифметичний зсув багатобайтних даних на один розряд вправо:

```
m1      db      100 dup (0aah)

        Clc             ;забезпечити 0 в старший біт (лінійний зсув)
;або записати в cf значення старшого біта старшого байта для арифметичного
; зсуву, наприклад
```

```
        Mov   al,m1+99
        Rcl   al,1

                    ; далі одинаково для лінійного та арифметичного зсувів
        Mov   si,100
@2:
        Rcr   m1[si],1
        Dec   si      ;ознака cf не змінюється (див. команду dec)
        Jnz   @2
                    ; після виконання зсуву в cf міститься
                    ; залишок від ділення числа на 2
```

Таблиця Д.53 – Команда ROL

ROL	**Циклічний зсув вліво**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
		*								*

Код операції	***Структура команди***	***Приклад на асемблері***	***Такти***
D0 /0	**ROL r/m8, 1**	**Rol cl, 1**	**1/3**
D2 /0	**ROL r/m8, CL**	**Rol dh, cl**	**4**
C0 /0 ib	**ROL r/m8, imm8**	**Rol ch, 3**	**1/3**
D1 /0	**ROL r/m16, 1**	**Rol bp, 1**	**1/3**
D3 /0	**ROL r/m16, CL**	**Rol ax, cl**	**4**
C1 /0 ib	**ROL r/m16, imm8**	**Rol cx, 7**	**1/3**
D1 /0	**ROL r/m32, 1**	**Rol ebx, 1**	**1/3**
D3 /0	**ROL r/m32, CL**	**Rol edx, 11**	**4**
C1 /0 ib	**ROL r/m32, imm8**	**Rol eax, 9**	**1/3**

Схема команди:	ROL *дані, кількість зсувів*

Призначення: виконання операції циклічного зсуву вліво.

Операнд *дані* може бути іменем 8, 16- або 32-розрядного регістра даних, або адресою пам’яті (зміщення в сегменті). Операнд *кількість* може бути константою або регістром CL.

Алгоритм роботи:

1) зсув всіх бітів операнда *дані* вліво на один розряд, при цьому старший біт операнда *дані* стає значенням молодшого біта цього ж операнда;

2) одночасно старший біт операнда *дані* стає значенням ознаки перенесення cf;
3) зазначені вище дві дії повторюються (*кількість* ***mod*** 8) разів для даних типу byte, (*кількість* ***mod*** 16) разів для даних типу word та (*кількість* ***mod*** 32) разів для даних типу dword.

Якщо операнд *дані* – адреса, то необхідно забезпечувати значення типу даних (кількість байтів 1, 2 чи 4), які зсуваються. Якщо дані зсуваються в регістрі CL (CX/ECX), тоді в якості лічильника зсувів використовується першопочатковий вміст CL. Якщо зсув більш ніж на один розряд, то ознака CF встановлюється за значенням останнього зсуву, а ознака OF є не визначеною. В операціях зсуву на один розряд значення старшого (знакового) розряду *після зсуву* дорівнює (OF ***xor*** CF).

Застосування: Команда ROL застосовується для обробки бітових полів. Наприклад, в системі команд процесорів сімейства 80x86 відсутній прямий доступ до старших розрядів регістрів даних. Команда rol дозволяє подолати цей недолік:

```
;поміняти місцями половинки регістра eax
   Rol    eax,16
   …      ;обробка старшої частини вмісту eax шляхом використання ax, ah та al

   Rol    eax,16      ;збереження обробленої старшої частини та
                      ;відновлення молодшої частини регістра eax
```

Таблиця Д.54 – Команда ROR

ROR	**Циклічний зсув вправо**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
		*								*

Код операції	***Структура команди***	***Приклад на асемблері***	***Такти***
1	**2**	**3**	**4**
D0 /1	**ROR r/m8, 1**	**Ror bl, 1**	**1/3**
D2 /1	**ROR r/m8, CL**	**Ror byte ptr [bx], cl**	**4**
C0 /1 ib	**ROR r/m8, imm8**	**Ror al, 4**	**1/3**

Продовження табл. Д.54

1	2	3	4
D1 /1	**ROR r/m16, 1**	**Ror di, 1**	**1/3**
D3 /1	**ROR r/m16, CL**	**Ror dx, cl**	**4**
C1 /1 ib	**ROR r/m16, imm8**	**Ror bx, 3**	**1/3**
D1 /1	**ROR r/m32, 1**	**Ror ecx, 1**	**1/3**
D3 /1	**ROR r/m32, CL**	**Ror eax, 20**	**4**
C1 /1 ib	**ROR r/m32, imm8**	**Ror ebp, 6**	**1/3**

Схема команди:	ROR *дані, кількість зсувів*

Призначення: виконання операції циклічного зсуву вправо.

Операнд *дані* може бути іменем 8, 16- або 32-розрядного регістра даних, або адресою пам’яті (зміщення в сегменті). Операнд *кількість* може бути константою або регістром CL.

Алгоритм роботи:

1) зсув всіх бітів операнда *дані* вправо на один розряд, при цьому молодший біт операнда *дані* стає значенням старшого біта цього ж операнда;
2) одночасно молодший біт операнда *дані* стає значенням ознаки перенесення cf;
3) зазначені вище дві дії повторюються (*кількість* ***mod*** 8) разів для даних типу byte, (*кількість* ***mod*** 16) разів для даних типу word та (*кількість* ***mod*** 32) разів для даних типу dword.

Якщо операнд *дані* – адреса, то необхідно забезпечувати значення типу даних (кількість байтів 1, 2 чи 4), які зсуваються. Якщо дані зсуваються в регістрі CL (CX/ECX), то в якості лічильника зсувів використовується першопочатковий вміст CL. Якщо зсув більш ніж на один розряд, то ознака CF встановлюється за значенням останнього зсуву,

а ознака OF є не визначеною. В операціях зсуву на один розряд значення OF визначається як результат операціїї ***xor*** між старшим (знаковим) розрядом та CF *до зсуву*.

Застосування: Команда ROR застосовується для обробки бітових полів. Важливим її застосування є забезпечення використання у реальному режимі можливостей 32-розрядної адресної арифметики при фактичному використанні лише 16 розрядів адрес, наприклад:

```
; нехай cx містить максимальний індекс масиву із слів за адресою Addr
    Ror    ecx,16
    Xor    cx,cx
    Ror    ecx,16
@1:

    Mov    ax,Addr[ecx*2]
    Loop   @1
```

Таблиця Д.55 – Префікси повторення

REP REPE REPZ REPNE REPNZ	**Префікс повторення**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**

Код операції	***Структура команди***	**Приклад на асемблері**	***Такти***
1	**2**	**3**	**4**
F3 A4	**REP MOVS m8, m8**	**Rep movsb**	**6*[3], 13*[4] , 13+(E)CX*[5]**
F3 A5	**REP MOVS m16, m16**	**Rep movsw**	**6*[3], 13*[4] ,13+(E)CX*[5]**
F3 A5	**REP MOVS m32, m32**	**Rep movsd**	**6*[3], 13*[4] , 13+(E)CX*[5]**
F2 AA	**REP STOS m8**	**Rep stosb**	**6*[3], 9+(E)CX*[6]**
F2 AB	**REP STOS m16**	**Rep stosw**	**6*[3], 9+(E)CX*[6]**
F2 AB	**REP STOS m32**	**Rep stosd**	**6*[3], 9+(E)CX*[6]**
F3 A6	**REPE CMPS m8, m8**	**Repe cmpsb**	**7*[3], 9+4*(E)CX*[6]**
F3 A7	**REPE CMPS m16,m16**	**Repe cmpsw**	**7*[3], 9+4*(E)CX*[6]**
F3 A7	**REPE CMPS m32,m32**	**Repe cmpsd**	**7*[3], 9+4*(E)CX*[6]**
F3 AE	**REPE SCAS m8**	**Repe scasb**	**7*[3], 9+4*(E)CX*[6]**
F3 AF	**REPE SCAS m16**	**Repe scasw**	**7*[3], 9+4*(E)CX*[6]**
F3 AF	**REPE SCAS m32**	**Repe scasd**	**7*[3], 9+4*(E)CX*[6]**

Продовження табл. Д.55

1	2	3	4
F2 A6	**REPNE CMPS m8, m8**	**Repne cmpsb**	**7*[3], 8+4*(E)CX*[6]**
F2 A7	**REPNE CMPS m16, m16**	**Repne cmpsw**	**7*[3], 8+4*(E)CX*[6]**
F2 A7	**REPNE CMPS m32, m32**	**Repne cmpsd**	**7*[3], 8+4*(E)CX*[6]**
F2 AE	**REPNE SCAS m8**	**Repne scasb**	**7*[3], 9+4*(E)CX*[6]**
F2 AF	**REPNE SCAS m16**	**Repne scasw**	**7*[3], 9+4*(E)CX*[6]**
F2 AF	**REPNE SCAS m32**	**Repne scasd**	**7*[3], 9+4*(E)CX*[6]**
***[3] якщо (E)CX = 0** ***[4] якщо (E)CX = 1** ***[5] якщо (E)CX > 1** ***[6] якщо (E)CX > 0**			

Призначення: виконання умовного і безумовного повторення наступної ланцюжкової команди (апаратний цикл на одну команду).

Алгоритм роботи:

В якості лічильника апаратного циклу використовується регістр cx при 16-розрядній адресації або регістр ecx при 32-розрядній адресації. Алгоритм роботи залежить від конкретного префікса та від наступної ланцюжкової команди.

Префікси REP, REPE і REPZ насправді мають однаковий код. Префікс rep використовується перед командами MOVS, STOS та їх еквівалентами з конкретним типом (розміром) даних.

Алгоритм для префікса REP:

1) Якщо cx/ecx=0, то передати управління команді, що розташована після даної ланцюжкової команди (вийти з циклу по rep);
2) Виконати ланцюжкову команду, cx/ecx=cx/ecx-1 та перейти до п.1.

Префікси REPE і REPZ використовуються перед ланцюжковими командами CMPS, SCAS та їх еквівалентами з конкретним типом (розміром) даних.

Алгоритм для префіксів REPE і REPZ:

1) Якщо cx/ecx=0, то передати управління команді, що розташована після даної ланцюжкової команди (вийти з циклу по repe і repz);
2) Виконати ланцюжкову команду, cx/ecx=cx/ecx-1;
3) Якщо (zf=1) і (cx/ecx ≠ 0) , то перейти до кроку 1, інакше вийти із циклу.

Префікси REPNE і REPNZ також мають однаковий код операції. Вони використовуються перед ланцюжковими командами CMPS, SCAS та їх еквівалентами з конкретним типом (розміром) даних

Алгоритм для префіксів REPNE і REPNZ:

1) Якщо cx/ecx=0, то передати управління команді, що розташована після даної ланцюжкової команди (вийти з циклу по REPNE і REPNZ);
2) Виконати ланцюжкову команду, cx/ecx=cx/ecx-1;
3) Якщо (zf=0) і (cx/ecx ≠ 0) , то перейти до кроку 1, інакше вийти із циклу.

Застосування: Префікси REP, REPE, REPZ, REPNE і REPNZ разом з ланцюжковими командами використовують для виконання наступних операцій над масивами (рядками, послідовністями) одно-, двох- чи чотирьохбайтних елементів:

a. пересилання масивів;
b. порівняння масивів;
c. пошук елемента в масиві;

d. заповнення масиву константою.

Таблиця Д.56 – Команди повернення RET

RET **RETN** **RETF**	**Повернення із процедури**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**

Код операції	***Структура команди***	***Опис типу повернення*** ***Приклад на асемблері***	***Такти***
C3	**RET**	**Внутрішньосегментне повернення** **Ret**	**2**
C3	**RETN**	**Внутрішньосегментне повернення** **Retn**	**2**
CB	**RET**	**Міжсегментне повернення** **Ret**	**4**
CB	**RETF**	**Міжсегментне повернення** **Retf**	**23**
C2 iw	**RET imm16**	**Внутрішньосегментне повернення та очистка стеку від параметрів** **Ret 6**	
C2 iw	**RETN imm16**	**Внутрішньосегментне повернення та очиститка стеку від параметрів** **Retn 2**	**3**
CA iw	**RET imm16**	**Міжсегментне повернення та очиститка стека від параметрів** **Ret 6**	**4**
CA iw	**RETF imm16**	**Міжсегментне повернення та очиститка стека від параметрів** **Retf 16**	**23**

Схема команди:	RET/RETN/RETF RET/RETN/RETF *число*

Операнд *число* є абсолютним значенням (константою).

Призначення: повернення управління із процедури.

Алгоритм роботи у реальному режимі**:**

1) Якщо внутрішньосегментне повернення, то
   a. Прочитати зі стеку два байти за логічною адресою ss:sp і записати їх в лічильник команд IP, EIP=EIP and 0ffffh;
   b. Збільшити SP на 2;
   c. Якщо команда ret має операнд *число* (код операції 0c2h), то sp=sp+*число*.
2) Якщо міжсегментне повернення, то
   a. Прочитати зі стеку два байта за логічною адресою ss:sp і записати їх в лічильник команд IP, EIP=EIP and 0ffffh;
   b. Прочитати зі стеку два байта за логічною адресою ss:sp+2 і записати їх в регістр CS;
   c. Збільшити SP на 4;
   d. Якщо команда ret має операнд *число* (код операції 0cah), то sp=sp+*число* .

Застосування: Команди RET/RETN/RETF зазвичай використовуються для повернення з процедур. Команди RETN/RETF явно задають тип повернення. Команда RET транслюється як команда ближнього (внутрішньосегментного) повернення в усіх випадках, крім того, коли вона розташована між директивами *proc far* та *endp*. Процедури мовою Асемблера рекомендується завжди оформляти за допомогою директив proc та endp. Рекомендується в процедурах використовувати лише команду RET та розміщати її безпосередньо перед директивою endp.

При міжмодульній взаємодії використовується ряд де-факто стандартизованих концепцій, в яких, між іншим, відповідальність за

очищення стеку від фактичних параметрів однозначно покладається на процедуру, яка була викликана (наприклад в концепції Паскаля). Реалізація такої концепції спрощується при наявності команди RET/RETN/RETF *число*, наприклад:

```
; нехай в цій точці програми sp містить 1000h
    Push   par_1        ;sp=0fffeh
    Push   par_2        ;sp=0fffch
    Call   proc1        ;sp=0fffah

    proc1  proc   near
    Ret           4     ;sp=1000h
    proc1  endp
```

У випадку програмування на C++, очищення стеку від параметрів виконує процедура, яка викликає:

```
    Push   par_1        ;sp=0fffeh
    Push   par_2        ;sp=0fffch
    Call   proc1        ;sp=0fffah
    Add    sp,4

    proc1  proc   near
    Ret                 ;sp=1000h
    proc1  endp
```

Таблиця Д.57 – Команда SAHF

SAHF	**Записати вміст AH в молодший байт регістра FLAGS**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
						*	*	*	*	*

Код операції	***Структура команди***	***Приклад на асемблері***	***Такти***
9E	**SAHF**	**Sahf**	**2**

Призначення: запис вмісту регістра ah у молодший байт регістра flags, в якому містяться п'ять ознак: cf, pf, af, zf і sf.

Алгоритм роботи: Команда завантажує молодший байт регістра flags вмістом регістра ah. У бітах 7, 6, 4, 2 і 0 регістри ah повинні, відповідно, міститися нові значення ознак sf, zf, af, pf і cf.

*Застосування: К*оманда використовується разом з командою LAHF. Через те, що регістр ознак безпосередньо недоступний, сполучення цих команд можна застосовувати для аналізу і, можливо, зміни стану деяких ознак у регістрі flags. Вміст старшої частини регістра ознак не змінюється.

```
;скинути в нуль ознаки cf, pf, af, zf i sf
Xor     ah,ah
Sahf
```

Таблиця Д.58 – Команди SAL, SHL

SAL **SHL**	**Арифметичний зсув вліво** **Лінійний зсув вліво**	**O** *	**D**	**I**	**T**	**S** *	**Z** *	**A** ?	**P** *	**C** *
Код операції	***Структура команди***	***Приклад на асемблері***						***Такти***		
D0 /4	**SAL r/m8, 1**	**Sal al, 1**						**1/3**		
D2 /4	**SAL r/m8, CL**	**Sal ah, cl**						**4**		
C0 /4 ib	**SAL r/m8, imm8**	**Sal dh, 4**						**1/3**		
D1 /4	**SAL r/m16, 1**	**Sal ax, 1**						**1/3**		
D3 /4	**SAL r/m16, CL**	**Sal ax, cl**						**4**		
C1 /4 ib	**SAL r/m16, imm8**	**Sal dx, 4**						**1/3**		
D1 /4	**SAL r/m32, 1**	**Sal eax, 1**						**1/3**		
D3 /4	**SAL r/m32, CL**	**Sal eax, cl**						**4**		
C1 /4 ib	**SAL r/m32, imm8**	**Sal edx, 4**						**1/3**		
D0 /4	**SHL r/m8, 1**	**Shl al, 1**						**1/3**		
D2 /4	**SHL r/m8, CL**	**Shl ah, cl**						**4**		
C0 /4 ib	**SHL r/m8, imm8**	**Shl dh, 4**						**1/3**		
D1 /4	**SHL r/m16, 1**	**Shl ax, 1**						**1/3**		
D3 /4	**SHL r/m16, CL**	**Shl ax, cl**						**4**		
C1 /4 ib	**SHL r/m16, imm8**	**Shl dx, 4**						**1/3**		
D1 /4	**SHL r/m32, 1**	**Shl eax, 1**						**1/3**		
D3 /4	**SHL r/m32, CL**	**Shl eax, cl**						**4**		
C1 /4 ib	**SHL r/m32, imm8**	**Shl edx, 4**						**1/3**		

Схема команди:	SAL/SHL *дані, кількість зсувів*

Призначення: виконання операції арифметичного/лінійного зсуву вліво.

Операнд *дані* може бути іменем 8, 16- або 32-розрядного регістра даних, або адресою пам'яті (зміщення в сегменті). Операнд *кількість_зсувів* може бути константою або регістром CL.

Алгоритм роботи:

1) зсув всіх розрядів вмісту операнда *дані* вліво на один розряд, при цьому старший розряд стає значенням ознаки перенесення cf, одночасно в молодший розряд записується нуль;
2) зазначені вище дві дії повторюються (*кількість_зсувів* ***mod*** 32) разів.

Якщо операнд *дані* – адреса, то необхідно забезпечувати значення типу даних (кількість байтів 1, 2 чи 4), які зсуваються. Якщо дані зсуваються в регістрі CL (CX/ECX), то в якості лічильника зсувів використовується першопочатковий вміст CL. Якщо зсув більш ніж на один розряд, то ознака CF встановлюється за значенням останнього зсуву, а ознака OF є не визначеною. В операціях зсуву на один розряд значення старшого (знакового) розряду *після зсуву* дорівнює (OF ***xor*** CF).

Застосування: Команди SAL/SHL використовується для зсуву розрядів операнда вліво. Фактично це одна й та ж сама команда процесора. Якщо вона застосовується в алгоритмах обробки бітових полів, то рекомендовується застосування мнемоніки SHL. Крім того, команду зручно застосовувати для швидкого множення операнда *дані* на $2^{(кількість\_зсувів\ \mathbf{mod}\ 32)}$. У цьому випадку, рекомендується використовувати мнемоніку sal, наприклад, помножити вміст eax на 1024:

```
Sal     eax,10
```

Таблиця Д.59 – Команда SAR

SAR	**Арифметичний зсув вправо**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
		*				*	*	?	*	*

Код операції	***Структура команди***	***Приклад на асемблері***	***Такти***
D0 /7	**SAR r/m8, 1**	**Sar al, 1**	**1/3**
D2 /7	**SAR r/m8, CL**	**Sar ah, cl**	**4**
C0 /7 ib	**SAR r/m8, imm8**	**Sar dh, 4**	**1/3**
D1 /7	**SAR r/m16, 1**	**Sar ax, 1**	**1/3**
D3 /7	**SAR r/m16, CL**	**Sar ah, cl**	**4**
C1 /7 ib	**SAR r/m16, imm8**	**Sar dx, 4**	**1/3**
D1 /7	**SAR r/m32, 1**	**Sar eax, 1**	**1/3**
D3 /7	**SAR r/m32, CL**	**Sar eax, cl**	**4**
C1 /7 ib	**SAR r/m32, imm8**	**Sar edx, 4**	**1/3**

Схема команди:	SAR *дані, кількість зсувів*

Призначення: виконання операції арифметичного зсуву вправо.

Операнд *дані* може бути іменем 8, 16- або 32-розрядного регістра даних, або адресою пам’яті (зміщення в сегменті). Операнд *кількість_зсувів* може бути константою або регістром CL.

Алгоритм роботи:

1) зсув всіх бітів вмісту операнда *дані* вправо на один розряд, при цьому молодший розряд стає значенням ознаки перенесення cf;
2) в старший (знаковий) розряд результату заноситься попереднє значення знакового розряду, тобто знак даних, які зсуваються, не змінюється;
3) зазначені вище дві дії повторюються (*кількість_зсувів* ***mod*** 32) разів.

Якщо операнд *дані* – адреса, то необхідно забезпечувати значення типу даних, які зсуваються (кількість байтів 1, 2 чи 4). Якщо дані зсуваються в регістрі CL (CX/ECX), то в якості лічильника зсувів

використовується першопочатковий вміст CL. Якщо зсув більш ніж на один розряд, то ознака CF встановлюється за значенням останнього зсуву, а ознака OF є не визначеною. В операціях зсуву на один розряд ознака OF скидається в 0.

Застосування: Команда SAR використовується для арифметичного зсуву розрядів операнда вправо. Фактично ця команда може застосовуватись для швидкого ділення 1, 2 та 4-х байтних чисел зі знаком на число $2^{(кількість\_зсувів \ \mathbf{mod}\ 32)}$. В результаті виконання команди в операнді *дані* буде знаходитися частка від ділення, яка буде мати той самий знак, що і ділене. При діленні на 2 (зсув на один розряд) ознака cf буде містити залишок від ділення на 2. Наприклад:

```
Mov   eax,-375679
                  ;(eax) розділити на 2
Sar   eax,1       ;eax=-187839 – частка, cf=1 – залишок
```

Таблиця Д.60 – Команда SHR

SHR	**Лінійний зсув вправо**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
		*				*	*	?	*	*
Код операції	***Структура команди***	***Приклад на асемблері***						***Такти***		
D0 /5	**SHR r/m8, 1**	**Shr al, 1**						**1/3**		
D2 /5	**SHR r/m8, CL**	**Shr ax, cl**						**4**		
C0 /5 ib	**SHR r/m8, imm8**	**Shr dh, 4**						**1/3**		
D1 /5	**SHR r/m16, 1**	**Shr ax, 1**						**1/3**		
D3 /5	**SHR r/m16, CL**	**Shr ax, cl**						**4**		
C1 /5 ib	**SHR r/m16, imm8**	**Shr dx, 4**						**1/3**		
D1 /5	**SHR r/m32, 1**	**Shr eax, 1**						**1/3**		
D3 /5	**SHR r/m32, CL**	**Shr eax, cl**						**4**		
C1 /5 ib	**SHR r/m32, imm8**	**Shr edx, 4**						**1/3**		

Схема команди:	SHR *дані*, *кількість зсувів*

Призначення: виконання операції лінійного зсуву вправо.

Операнд *дані* може бути іменем 8, 16- або 32-розрядного регістра даних, або адресою пам'яті (зміщення в сегменті). Операнд *кількість_зсувів* може бути константою або регістром CL.

Алгоритм роботи:

1) зсув всіх розрядів вмісту операнда *дані* вправо на один розряд, при цьому молодший розряд стає значенням ознаки перенесення cf; одночасно в старший розряд записується нуль;
2) зазначені вище дві дії повторюються (*кількість_зсувів* ***mod*** 32) разів.

Якщо операнд *дані* – адреса, то необхідно забезпечувати значення типу даних (кількість байтів 1, 2 чи 4), які зсуваються. Якщо дані зсуваються в регістрі CL (CX/ECX), то в якості лічильника зсувів використовується першопочатковий вміст CL. Якщо зсув більш ніж на один розряд, то ознака CF встановлюється за значенням останнього зсуву, а ознака OF є не визначеною. В операціях зсуву на один розряд значення старшого розряду *до зсуву* записується в OF.

Застосування: Команда SHR використовується в алгоритмах обробки бітових полів для лінійного зсуву розрядів операнда вправо. Крім того, ця команда може використовуватись для швидкого ділення 1, 2 та 4-х байтних беззнакових чисел на число $2^{(кількість\_зсувів\ \mathbf{mod}\ 32)}$. В результаті виконання команди в операнді *дані* буде знаходиться частка від ділення. При діленні на 2 (зсув на один розряд) ознака cf буде містити залишок від ділення на 2.

```
Mov   cl,4
Shr   eax,cl        ;(eax) розділити на 16
```

Таблиця Д.61 – Команда SBB

SBB	**Віднімання цілих чисел з позикою**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
		*				*	*	*	*	*

Код операції	***Структура команди***	***Приклад на асемблері***	***Такти***
1C ib	**SBB AL, imm8**	**Sbb al, 0AAh**	**1**
1D iw	**SBB AX, imm16**	**Sbb ax, 0FODh**	**1**
1D id	**SBB EAX, imm32**	**Sbb eax, 23456789h**	**1**
80 /3 ib	**SBB r/m8, imm8**	**Sbb ah, 5**	**1/3**
81 /3 iw	**SBB r/m16, imm16**	**Sbb dx,0DBBh**	**1/3**
81 /3 id	**SBB r/m32, imm32**	**Sbb edx, 0CAAAAh**	**1/3**
83 /3 ib	**SBB r/m16, imm8**	**Sbb cx, 0AAh**	**1/3**
83 /3 ib	**SBB r/m32, imm8**	**Sbb ecx, 02h**	**1/3**
18 /r	**SBB r/m8, r8**	**Sbb [di], ah**	**1/3**
19 /r	**SBB r/m16, r16**	**Sbb bx, si**	**1/3**
19 /r	**SBB r/m32, r32**	**Sbb memory, eax**	**1/3**
1A /r	**SBB r8, r/m8**	**Sbb dl, sum**	**1/2**
1B /r	**SBB r16, r/m16**	**Sbb di, [si+12]**	**1/2**
1B /r	**SBB r32, r/m32**	**Sbb ecx,raznost**	**1/2**

Схема команди:	SBB *приймач, джерело*

Призначення: віднімання цілих чисел з урахуванням результату попереднього віднімання командами SBB і SUB (за станом ознаки перенесення cf).

Алгоритм роботи:

1) *приймач=приймач-джерело*-cf;
2) встановити ознаки cf, pf, af, zf, sf та of за результатом віднімання.

Цілі зі знаком задаються у доповняльному коді. Результат віднімання не залежить від типу вхідних даних (числа зі знаком чи без знаку). Трактування результату при необхідності здійснюється шляхом аналізу ознак.

Співвідношення розміру операндів – див. зауваження до команди ADC.

Застосування: Команда SBB використовується для виконання віднімання багатобайтних цілих чисел з урахуванням можливої позики із молодших розрядів, наприклад:

```
;виконати віднімання 64-бітних цілих без знаку: val_1-val_2
val_1  dq    2 dup (1000000h)
val_2  dq    2 dup (0fffffh)
rez    dq    2 dup (?)

Mov    eax,dword ptr val_1      ; eax=0000h
Sub    eax,dword ptr val_2      ; eax=0000h-0ffffh=0001h, cf=1
Mov    dword ptr rez,eax        ;молодша частина результату
Mov    eax,dword ptr val_1+4    ; eax=0100h
Sbb    eax,dword ptr val_2+4    ; eax=0100h-0fh-1=0f0h
Mov    dword ptr rez+4,eax      ;старша частина результату 0f00001h
```

Таблиця Д.62 – Команди SCAS

SCAS **SCASB** **SCASW** **SCASD**	**Порівняти елемент рядка з елементом в регістрі al/ax/eax**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
		*				*	*	*	*	*

Код операції	***Структура команди***	***Приклад на асемблері***	***Такти***
AE	**SCAS m8**	**Scas byte ptr [di]**	**4**
AF	**SCAS m16**	**Scas word ptr [di]**	**4**
AF	**SCAS m32**	**Scas dword ptr [di]**	**4**
AE	**SCASB**	**Scasb**	**4**
AF	**SCASW**	**Scasw**	**4**
AF	**SCASD**	**Scasd**	**4**

Схема команди:	SCAS *приймач* SCASB SCASW SCASD

Призначення: пошук елемента в рядку (в масиві, в послідовності елементів, в ланцюжку).

Алгоритм роботи:

1) Від вмісту регістра eax/ax/al відняти значення операнда *приймач* (елемента рядка розміром байт, слово або подвійне слово). Логічна адреса елемента рядка задається парою регістрів es:edi/di. Заміна сегментного регістра es не допускається.
2) За результатом віднімання встановити ознаки.
3) В залежності від стану ознаки df змінити значення регістра edi/di:
    a. Якщо df=0, то збільшити вміст цього регістра на довжину елемента послідовності в байтах;
    b. Якщо df=1, то зменшити вміст цього регістра на довжину елемента послідовності в байтах.

Команди SCASW та SCASD мають один і той самий код операції. Тому при трансляції команд SCASW та SCASD Асемблер генерує префікс зміни розрядності даних (код 66h), якщо розрядність даних, яка задана в директиві Segment, не відповідає мнемокоду команди. В Асемблері відсутні засоби для автоматичного генерування префікса зміни розрядності адрес для команд SCASB, SCASW та SCASD (відповідно використання регістра DI чи EDI).

Машинного аналога для команди SCAS немає. При трансляції команди SCAS Асемблер, з'ясувавши тип операнда, генерує відповідний код операції та може автоматично згенерувати префікс зміни розрядності даних (код 66h), а з'ясувавши розрядність зміщення, може автоматично згенерувати префікс зміни розрядності адрес (код 67h).

ЗАУВАЖЕННЯ. У реальному режимі при використанні регістра EDI старші 16 розрядів 32-розрядного зміщення ігноруються.

Застосування: Команди сканування разом з префіксом repne використовуються для пошуку елемента в послідовності (рядку, масиві).

або пропуску однакових елементів послідовності (рядка, масиву). Наприклад, підрахуємо кількість букв “e” у деякому рядку:

```
line1     db    ‘a.....z’

      Mov   cx,length line1     ;cx-довжина рядка
      Mov   ax,ds
      Mov   es,ax
      Mov   di,offset line1
      Xor   dx,dx               ;dx-лічильник кількості букв ‘e’
      Mov   al,’e’
@10:
      Repne scasb
      Jcxz  @20
      Inc   dx
      Jmp   @10
@20:
      Jz    @30
      Inc   dx                  ;якщо останній символ рядка – буква ‘e’
@30:
```

Команди сканування разом з префіксом REPE використовуються для пропуску однакових елементів послідовності (рядка, масиву). Наприклад, пропуску пробілів:

```
; нехай в в DI міститься зміщення першого пробілу в рядку
; вважємо, що DF=0
  Mov   al,’ ‘
  Repe  scasb
  Dec   di          ; DI містить зміщення першого не пробілу
```

Таблиця Д.63 – Команда STC

STC	**Встановити ознаку перенесення**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
										1

Код операції	***Структура команди***	***Приклад на асемблері***	***Такти***
F9	**STC**	**Stc**	**2**

Призначення: встановлення ознаки перенесення cf у 1.

Алгоритм роботи: cf=1.

Застосування: Необхідність встановлення ознаки перенесення cf у 1 може виникнути при роботі з командами зсуву, командами SBB і ADC та

ін. Наявність в системі команд цієї команди поряд з командою CLC (скидання cf у 0), а також команд JNC та JC обумовлює зручність використання ознаки cf для зберігання однобітного фактичного параметру в асемблерних процедурах.

Таблиця Д.64– Команда STD

STD	**Встановити ознаку напрямку**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
			1							

Код операції	***Структура команди***	***Приклад на асемблері***	***Такти***
FD	**STD**	**Std**	**2**

Призначення: встановлення ознаки напрямку df у 1.

Алгоритм роботи: df=1.

Застосування: Ця команда використовується при роботі з ланцюговими командами. Одиничний стан ознаки df змушує мікропроцесор робити декремент вмісту регістрів si/esi і di/edi при виконанні ланцюгових операцій, тобто проводити перегляд рядків (масивів, послідовностей) у зворотньому напрямку.

Таблиця Д.65– Команда STI

STI	**Встановити ознаку переривання**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
				1						

Код операції	***Структура команди***	***Приклад на асемблері***	***Такти***
FB	**STI**	**Sti**	**7**

Призначення: дозволити переривання від зовнішніх пристроїв.

Алгоритм роботи: if=1.

Застосування: Дана команда використовується для установки ознаки if в 1. Така необхідність може виникнути при розробці програм обробки переривань. При передачі управління процедурі обробки переривань від зовнішніх пристроїв, процесор автоматично скидає ознаку if в 0, забороняючи тим самим переривання від інших зовнішніх пристроїв. Якщо

в ЕОМ існують зовнішні пристрої, які потребують негайного втручання, то для забезпечення переривання процедури обробки переривань необхідно, при можливості, якомога скоріше виконати команду STI.

Крім того, в будь-яких програмах, особливо в програмах, які обслуговують зовнішні пристрої, можуть бути критичні щодо швидкодії ділянки. Переривання від інших зовнішніх пристроїв на таких ділянках може призвести до аварійних ситуацій. Тому на початку таких ділянок виконують команду CLI (if=0), а по закінченню – команду STI.

Таблиця Д.66 – Команда STOS

STOS **STOSB** **STOSW** **STOSD**	**Зберегти елемент рядка**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**

Код операції	***Структура команди***	***Приклад на асемблері***	***Такти***
AA	**STOS m8**	**Stos byte ptr dst**	**3**
AB	**STOS m16**	**Stos word ptr dst**	**3**
AB	**STOS m32**	**Stos dword ptr dst**	**3**
AA	**STOSB**	**Stosb**	**3**
AB	**STOSW**	**Stosw**	**3**
AB	**STOSD**	**Stosd**	**3**

Схема команди:	STOS *приймач* STOSB STOSW STOSD

Призначення: збереження елемента з регістра-акумулятора al/ax/eax в рядку (в масиві, в послідовності елементів, в ланцюжку).

Алгоритм роботи:

1) Вміст регістра eax/ax/al записати в приймач (елемент рядка або масиву розміром байт, слово або подвійне слово). Логічна адреса

елемента рядка задається парою регістрів es:edi/di. Заміна сегментного регістра es не допускається.

2) В залежності від стану ознаки df змінити значення регістра edi/di:
   a. Якщо df=0, то збільшити вміст цього регістра на довжину елемента послідовності в байтах;
   b. Якщо df=1, то зменшити вміст цього регістра на довжину елемента послідовності в байтах.

Команди stosw та stosd мають один і той самий код операції. Тому при трансляції команд STOSW та STOSD Асемблер генерує префікс зміни розрядності даних (код 66h), якщо розрядність даних, яка задана в директиві Segment, не відповідає мнемокоду команди. В Асемблері відсутні засоби для автоматичного генерування префікса зміни розрядності адрес для команд SCASB, SCASW та SCASD (відповідно використання регістра di чи edi).

Машинного аналога для команди STOS немає. При трансляції команди stos Асемблер, з'ясувавши тип операнда, генерує відповідний код операції та може автоматично згенерувати префікс зміни розрядності даних (код 66h), а з'ясувавши розрядність зміщення в сенменті, може автоматично згенерувати префікс зміни розрядності адрес (код 67h).

ЗАУВАЖЕННЯ. У реальному режимі при використанні регістра EDI старші 16 розрядів 32-розрядного зміщення ігноруються.

Застосування: Команди зберігають елемент рядка із регістрів al/ax/eax в комірці (комірках) пам'яті. Перед командою stos можна вказати префікс повторення REP, у цьому випадку з'являється можливість заповнити блок пам'яті константою, наприклад, очистити сегмент пам'яті:

```
Mov   cx,03fffh
Xor   eax,eax
Xor   di,di
```

```
Cld
Rep   Stosd                ;es – адреса сегмента пам’яті
```

Таблиця Д.67 – Команда SUB

SUB	**Віднімання цілих чисел**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
		*				*	*	*	*	*
Код операції	***Структура команди***	***Приклад на асемблері***						***Такти***		
2C ib	**SUB AL, imm8**	**Sub al, 0AAh**						**1**		
2D iw	**SUB AX, imm16**	**Sub ax, 0FODh**						**1**		
2D id	**SUB EAX, imm32**	**Sub eax, 23456789h**						**1**		
80 /5 ib	**SUB r/m8, imm8**	**Sub ah, 5**						**1/3**		
81 /5 iw	**SUB r/m16, imm16**	**Sub dx,0DBBh**						**1/3**		
81 /5 id	**SUB r/m32, imm32**	**Sub edx, 0CAAAAh**						**1/3**		
83 /5 ib	**SUB r/m16, imm8**	**Sub cx, 0AAh**						**1/3**		
83 /5 ib	**SUB r/m32, imm8**	**Sub ecx, 02h**						**1/3**		
28 /r	**SUB r/m8, r8**	**Sub [di], ah**						**1/3**		
29 /r	**SUB r/m16, r16**	**Sub bx, si**						**1/3**		
29 /r	**SUB r/m32, r32**	**Sub memory, eax**						**1/3**		
2A /r	**SUB r8, r/m8**	**Sub dl, sum**						**1/2**		
2B /r	**SUB r16, r/m16**	**Sub di, [si+12]**						**1/2**		
2B /r	**SUB r32, r/m32**	**Sub ecx,raznost**						**1/2**		

Схема команди:	SUB *приймач, джерело*

Призначення: Віднімання цілих чисел.

Алгоритм роботи:

1) *приймач=приймач-джерело*;
2) встановити ознаки cf, pf, af, zf, sf та of за результатом віднімання.

Цілі зі знаком задаються у доповняльному коді. Результат віднімання не залежить від типу вхідних даних (числа зі знаком чи без знаку). Трактування результату при необхідності здійснюється шляхом аналізу ознак. Співвідношення розміру операндів – згідно зауваження до команди adc.

Застосування: Команда SUB використовується для віднімання цілочисельних операндів або для віднімання молодших частин значень багатобайтних операндів (див. команду SBB).

Таблиця Д.68 – Команда TEST

TEST	**Порозрядне тестування**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
		*				*	*	*	*	*

Код операції	***Структура команди***	***Приклад на асемблері***	***Такти***
A8 ib	**TEST AL, imm8**	**Test al, 0AAh**	**1**
A9 iw	**TEST AX, imm16**	**Test ax, 0FODh**	**1**
A9 id	**TEST EAX, imm32**	**Test eax, 2456789h**	**1**
F6 /0 ib	**TEST r/m8, imm8**	**Test ah, 5**	**1/2**
F7 /0 iw	**TEST r/m16, imm16**	**Test dx,0DBBh**	**1/2**
F7 /0 id	**TEST r/m32, imm32**	**Test edx, 0CAAAAh**	**1/2**
84 /r	**TEST r/m8, r8**	**Test [di], ah**	**1/2**
85 /r	**TEST r/m16, r16**	**Test bx, si**	**1/2**
85 /r	**TEST r/m32, r32**	**Test memory, eax**	**1/2**

Схема команди:	TEST *приймач, джерело*

Призначення: виконання порозрядної операції AND операндів *приймач* і *джерело* розмірністю байт, слово або подвійне слово.

Алгоритм роботи:

1) Виконати операцію логічного множення над операндами *приймач* і *джерело*: біт результату дорівнює 1, якщо відповідні біти операндів рівні 1, в інших випадках біт результату дорівнює 0. Результат операції нікуди не записується.
2) за результатом операції встановити ознаки sf,zf та pf. Ознаки of та cf скидається в 0.

Застосування: Команду зручно використовувати для одержання інформації про стан заданих бітів операнда *приймач*. Для аналізу

результату використовується ознака zf, що дорівнює 1, якщо результат логічного множення дорівнює нулю, наприклад:

```
Test   al,4h
Jnz    m1          ;перехід, якщо другий біт al дорівнює 1
```

Таблиця Д.69 – Команда XCHG

XCHG	**Обмін**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
		*				*	*	*	*	*

Код операції	***Структура команди***	***Приклад на асемблері***	***Такти***
90+ rw	**XCHG AX, r16**	**Xchg ax, bx**	**2**
90+ rw	**XCHG r16, AX**	**Xchg bx, ax**	**2**
90+ rd	**XCHG EAX, r32**	**Xchg eax, ebx**	**2**
90+ rd	**XCHG r32, EAX**	**Xchg edi, eax**	**2**
86 /r	**XCHG r/m8, r8**	**Xchg dh, dl**	**3**
86 /r	**XCHG r8, r/m8**	**Xchg cl, [bx]**	**3**
87 /r	**XCHG r/m16, r16**	**Xchg [di], dx**	**3**
87 /r	**XCHG r16, r/m16**	**Xchg ax, [bp]**	**3**
87 /r	**XCHG r/m32, r32**	**Xchg ecx, edx**	**3**
87 /r	**XCHG r32, r/m32**	**Xchg edi, [esi+5]**	**3**

Схема команди:	XCHG *приймач, джерело*

Призначення: обмін даними між регістрами або між регістром і пам'яттю.

Алгоритм роботи:

1) *temp:=приймач;*
2) *приймач:= джерело;*
3) *джерело:=temp.*

Застосування: Практично кожний із регістрів загального призначення в процесорах 80x86 має властиву тільки йому спеціалізацію. Наприклад, тільки регістр CX використовується як лічильник в командах зсуву та

циклах. Команду XCHG зручно використовувати, коли особливі можливості того чи іншого регістру необхідні багаторазово й одночасно. Наприклад, в циклі, який реалізується за допомогою команди LOOP, необхідно виконувати команди зсуву зі змінною кількістю зсувів:

```
      kshl   db     ?       ; змінна, в якій зберігається результат
                             ; обчислення кількості зсувів
      Mov    cx,30   ;
  @1:                ; початок тіла циклу
      Xchg   cl,kshl
      Shl    ax,cl
      Xchg   cl,kshl
      Loop   @1
```

Таблиця Д.70 – Команда XOR

XOR	**Логічне “Виключне АБО”**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
		*				*	*	*	*	*

Код операції	***Структура команди***	***Приклад на асемблері***	***Такти***
34 ib	**XOR AL, imm8**	**Xor al, 0AAh**	**1**
35 iw	**XOR AX, imm16**	**Xor ax, 0FODh**	**1**
35 id	**XOR EAX, imm32**	**Xor eax, 23456789h**	**1**
80 /6 ib	**XOR r/m8, imm8**	**Xor byte ptr [di], 5**	**1/3**
81 /6 iw	**XOR r/m16, imm16**	**Xor dx,0DBBh**	**1/3**
81 /6 id	**XOR r/m32, imm32**	**Xor edx,0CAAAAh**	**1/3**
83 /6 ib	**XOR r/m16, imm8**	**Xor cx, 0AAh**	**1/3**
83 /6 ib	**XOR r/m32, imm8**	**Xor ecx, 02h**	**1/3**
30 /r	**XOR r/m8, r8**	**Xor [di], ah**	**1/3**
31 /r	**XOR r/m16, r16**	**Xor bx, si**	**1/3**
31 /r	**XOR r/m32, r32**	**Xor memory, eax**	**1/3**
32 /r	**XOR r8, r/m8**	**Xor dl,sum**	**1/2**
33 /r	**XOR r16, r/m16**	**Xor di, [si+12]**	**1/2**
33 /r	**XOR r32, r/m32**	**Xor ecx,raznost**	**1/2**

Схема команди:	XOR *приймач, джерело*

Призначення: виконання порозрядної логічної операції “Виключне АБО”.

Алгоритм роботи:

1) кожен біт результату дорівнює 0, якщо відповідні біти обох операндів дорівнюють 0 або відповідні біти обох операндів дорівнюють 1, в інших випадках біт результату дорівнює 1 (порозрядна сума за ***mod*** 2);
2) записати результат операції в *приймач*;

Співвідношення розміру операндів – згідно зауваження до команди adc.

Застосування: Команду зручно використовувати для інвертування або порівняння визначених бітів операндів.

```
                                ; порозрядна інверсія операндів
Mov   al,01110011b
Mov   ah,0ffh
Xor   al,ah                     ;ah=10001100b – інверсія бітів al,
; цю команду часто використовують для занесення 0 в регістр, наприклад
Xor   eax,eax
```